W3School Scala 教程

wizardforcel

Published with GitBook

目錄

W3School Scala 教程

作者：W3School

来源：Scala 教程

Scala 教程

Scala 是一门多范式（multi-paradigm）的编程语言，设计初衷是要集成面向对象编程和函数式编程的各种特性。

Scala 运行在Java虚拟机上，并兼容现有的Java程序。

Scala 源代码被编译成Java字节码，所以它可以运行于JVM之上，并可以调用现有的Java类库。

谁适合阅读本教程？

本教程适合想从零开始学习 Scala 编程语言的开发人员。当然本教程也会对一些模块进行深入，让你更好的了解 Scala 的应用。

学习本教程前你需要了解

在继续本教程之前，你应该了解一些基本的计算机编程术语。如果你学习过Java编程语言，将有助于你更快的了解 Scala 编程。

学习 Java 教程。

第一个 Scala 程序：Hello World

以下是用 Scala 编写的典型 Hello World 程序：

实例（HelloWorld.scala）

```
object HelloWorld {
    def main(args: Array[String]): Unit = {
        println("Hello, world!")
    }
}
```

运行实例 ?

将以上代码保存为 HelloWorld.scala 文件，执行以上 scala 程序（你也可以直接在线执行）：

```
$ scalac HelloWorld.scala
$ scala HelloWorld.scala
```

输出结果为：

```
Hello, world!
```

相关文档推荐

以下是一份 Scala语言规范.pdf 文档，可作为学习参考：

Scala 简介

Scala 是 Scalable Language 的简写，是一门多范式的编程语言

联邦理工学院洛桑（EPFL）的Martin Odersky于2001年基于Funnel的工作开始设计Scala。

Funnel是把函数式编程思想和Petri网相结合的一种编程语言。

Odersky先前的工作是Generic Java和javac（Sun Java编译器）。Java平台的Scala于2003年底/2004年初发布。.NET平台的Scala发布于2004年6月。该语言第二个版本，v2.0，发布于2006年3月。

截至2009年9月，最新版本是版本2.7.6 。Scala 2.8预计的特性包括重写的Scala类库（Scala collections library）、方法的命名参数和默认参数、包对象（package object），以及Continuation。

2009年4月，Twitter宣布他们已经把大部分后端程序从Ruby迁移到Scala，其余部分也打算要迁移。此外， Wattzon已经公开宣称，其整个平台都已经是基于Scala基础设施编写的。

Scala 特性

面向对象特性

Scala是一种纯面向对象的语言，每个值都是对象。对象的数据类型以及行为由类和特质描述。

类抽象机制的扩展有两种途径：一种途径是子类继承，另一种途径是灵活的混入机制。这两种途径能避免多重继承的种种问题。

函数式编程

Scala也是一种函数式语言，其函数也能当成值来使用。Scala提供了轻量级的语法用以定义匿名函数，支持高阶函数，允许嵌套多层函数，并支持柯里化。Scala的case class及其内置的模式匹配相当于函数式编程语言中常用的代数类型。

更进一步，程序员可以利用Scala的模式匹配，编写类似正则表达式的代码处理XML数据。

静态类型

Scala具备类型系统，通过编译时检查，保证代码的安全性和一致性。类型系统具体支持以下特性：

- 泛型类
- 协变和逆变
- 标注
- 类型参数的上下限约束
- 把类别和抽象类型作为对象成员
- 复合类型
- 引用自己时显式指定类型
- 视图
- 多态方法

扩展性

Scala的设计秉承一项事实，即在实践中，某个领域特定的应用程序开发往往需要特定于该领域的语言扩展。Scala提供了许多独特的语言机制，可以以库的形式轻易无缝添加新的语言结构：

- 任何方法可用作前缀或后缀操作符
- 可以根据预期类型自动构造闭包。

并发性

Scala使用Actor作为其并发模型，Actor是类似线程的实体，通过邮箱发收消息。Actor可以复用线程，因此可以在程序中可以使用数百万个Actor,而线程只能创建数千个。在2.10之后的版本中，使用Akka作为其默认Actor实现。

谁使用了 Scala

- 2009年4月，Twitter宣布他们已经把大部分后端程序从Ruby迁移到Scala，其余部分也打算要迁移。
- 此外，Wattzon已经公开宣称，其整个平台都已经是基于Scala基础设施编写的。
- 瑞银集团把Scala用于一般产品中。
- Coursera把Scala作为服务器语言使用。

Scala Web 框架

以下列出了两个目前比较流行的 Scala 的 Web应用框架：

- Lift 框架
- Play 框架

Scala 安装

Scala 语言可以运行在Window、Linux、Unix、 Mac OS X等系统上。

Scala是基于java之上，大量使用java的类库和变量，必须使用Scala之前必须先安装 Java（>1.5版本）。

Mac OS X 和 Linux 上安装 Scala

第一步：Java 设置

确保你本地以及安装了 JDK 1.5 以上版本，并且设置了 JAVA_HOME 环境变量及 JDK 的bin 目录。

我们可以使用以下命令查看是否安装了 Java：

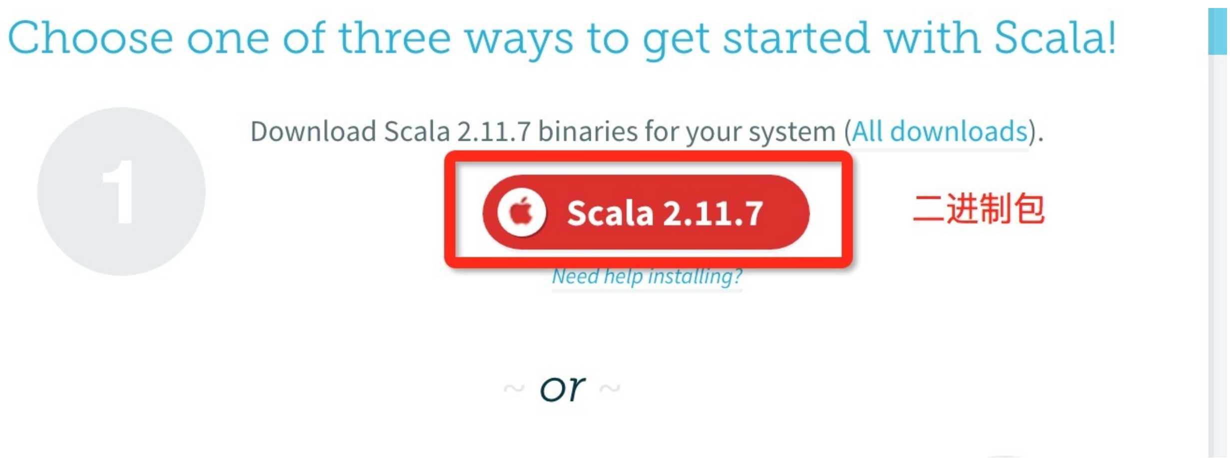
```
$ java -version
java version "1.8.0_31"
Java(TM) SE Runtime Environment (build 1.8.0_31-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.31-b07, mixed mode)
$
```

接着，我们可以查看是否安装了 Java 编译器。输入以下命令查看：

```
$ javac -version
javac 1.8.0_31 $
```

如果还为安装，可以参考我们的Java 开发环境配置。

接下来，我们可以从 Scala 官网地址 http://www.scala-lang.org/downloads 下载 Scala 二进制包，本教程我们将下载 *2.11.7*版本，如下图所示：



解压缩文件包，可将其移动至/usr/local/share下：

```
mv scala-2.11.7 scala                   # 重命名 Scala 目录
mv /download/scalapath /usr/local/share # 下载目录需要按你实际的下载路径
```

修改环境变量，如果不是管理员可使用 sudo 进入管理员权限，修改配置文件profile:

```
vim /etc/profile

或

sudo vim /etc/profile
```

在文件的末尾加入:

```
export PATH="$PATH:/usr/local/share/scala/bin"
```

:wq!保存退出，重启终端，执行 scala 命令，输出以下信息，表示安装成功：

```
$ scala
Welcome to Scala version 2.11.7 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_31).
Type in expressions to have them evaluated.
Type :help for more information.
```

> 注意：在编译的时候，如果有中文会出现乱码现象，解决方法查看：Scala 中文乱码解决

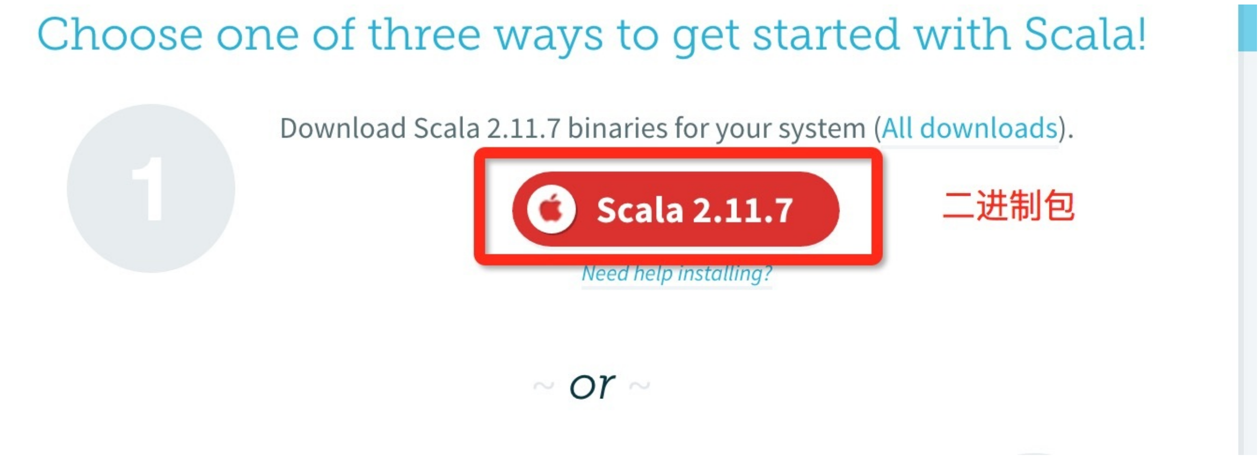
window 上安装 Scala

第一步：Java 设置

检测方法前文已说明，这里不再描述。

如果还为安装，可以参考我们的Java 开发环境配置。

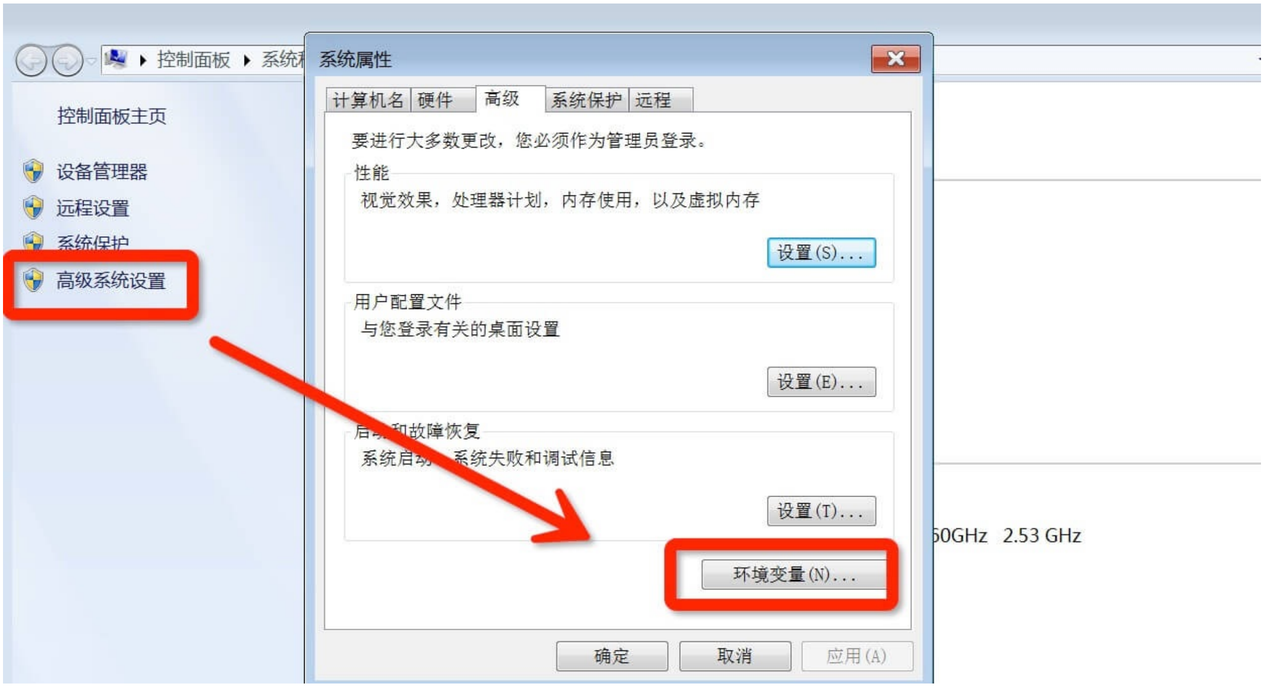
接下来，我们可以从 Scala 官网地址 http://www.scala-lang.org/downloads 下载 Scala 二进制包，本教程我们将下载 *2.11.7*版本，如下图所示：



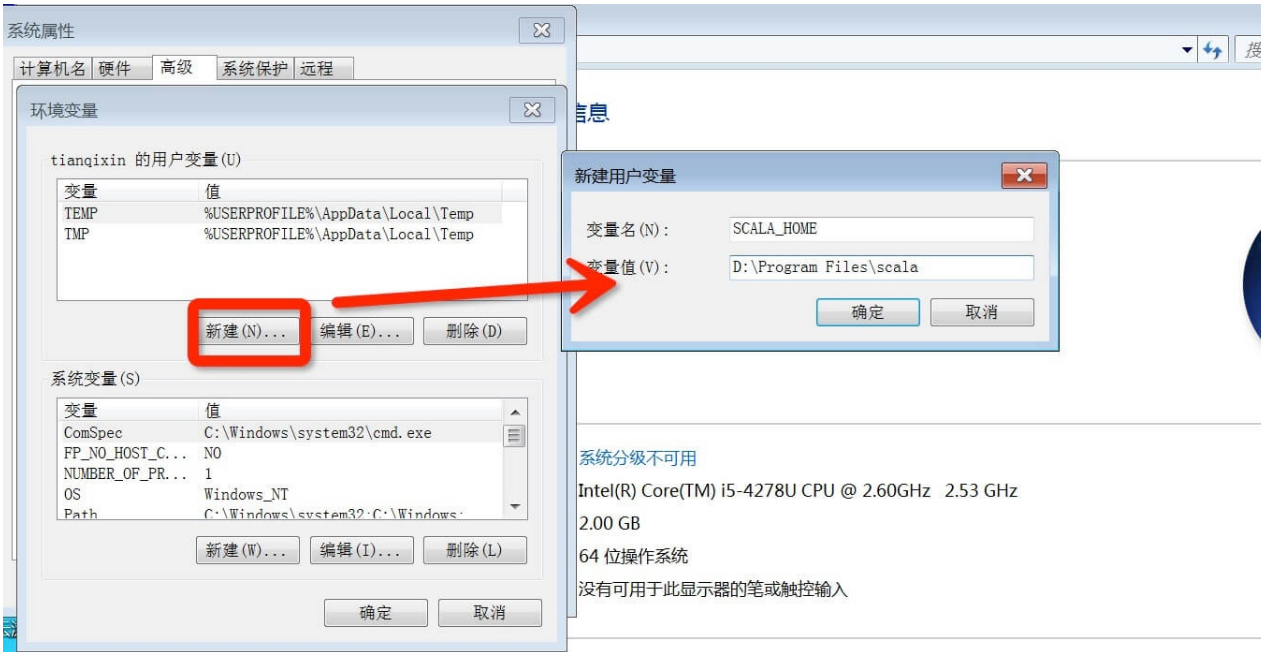
下载后，双击 msi 文件，一步步安装即可，安装过程你可以使用默认的安装目录。

安装好scala后，系统会自动提示，单击 finish，完成安装。

右击我的电脑，单击"属性"，进入如图所示页面。下面开始配置环境变量，右击【我的电脑】--【属性】--【高级系统设置】--【环境变量】，如图：

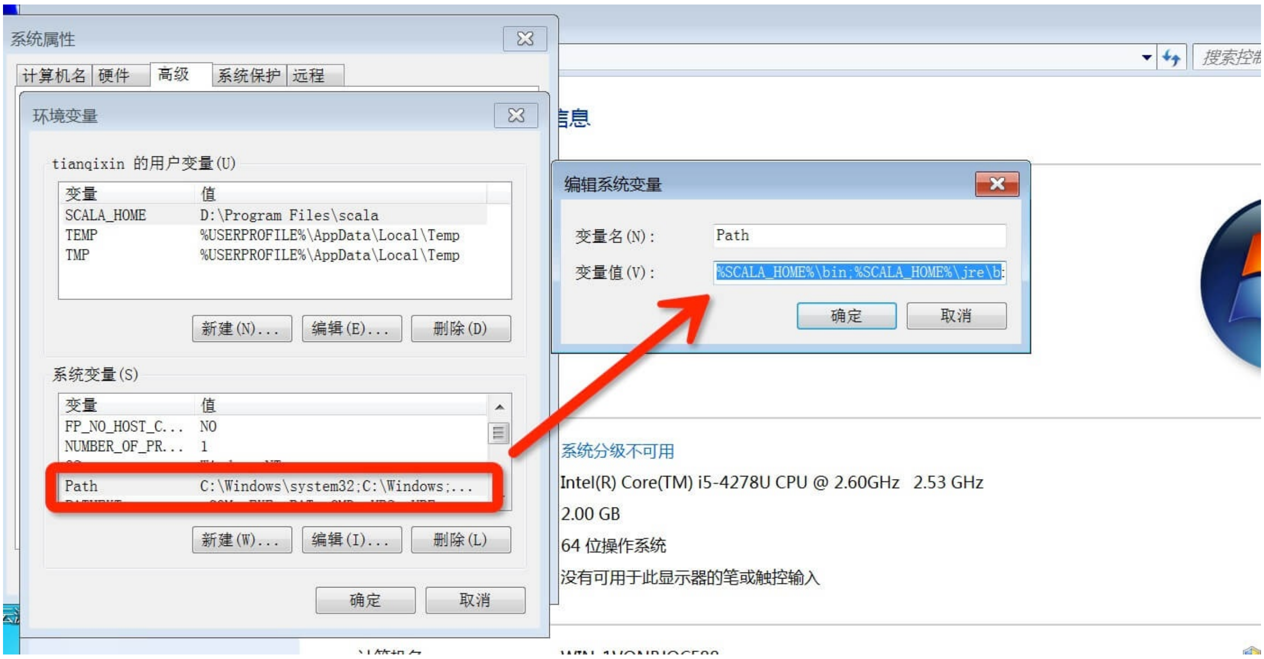


设置 SCALA_HOME 变量：单击新建，在变量名栏输入：**SCALA_HOME**: 变量值一栏输入：**D:\Program Files\scala** 也就是scala的安装目录，根据个人情况有所不同，如果安装在C盘，将"D"改成"C"即可。



设置 Path 变量：找到系统变量下的"Path"如图，单击编辑。在"变量值"一栏的最前面添加如下的路径： %SCALA_HOME%\bin;%SCALA_HOME%\jre\bin;

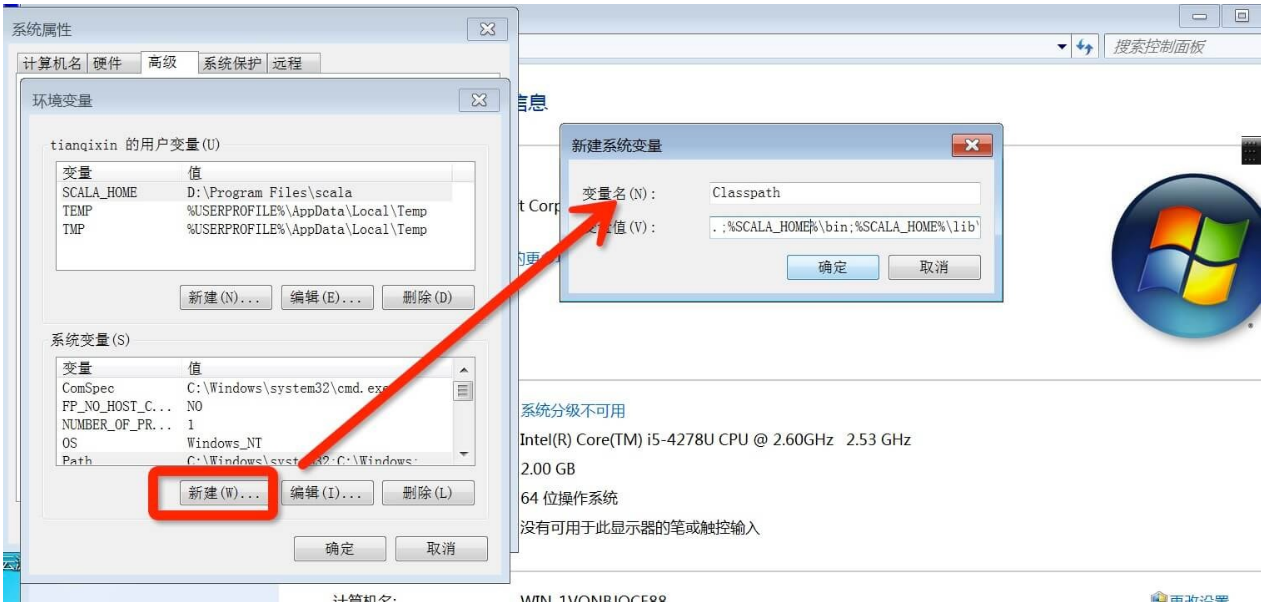
注意：后面的分号 ； 不要漏掉。



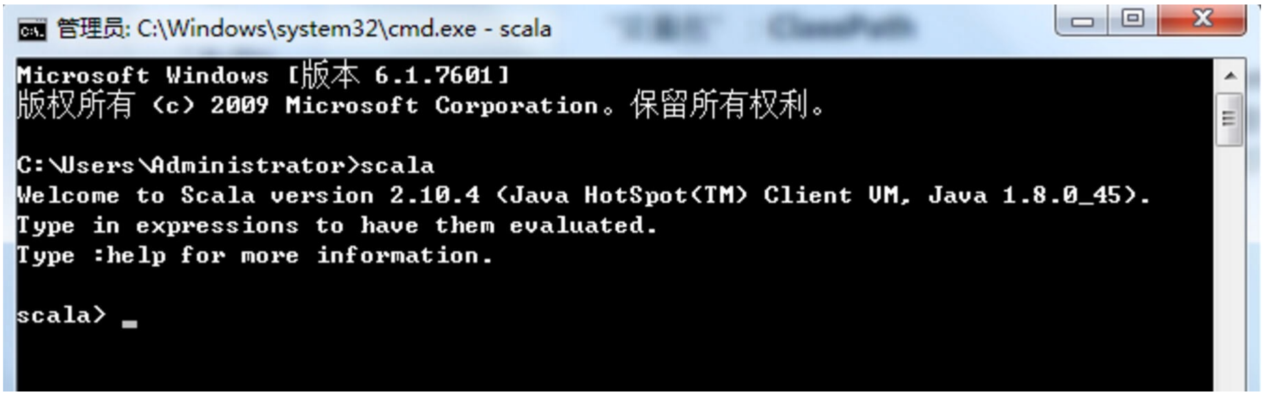
设置 Classpath 变量：找到找到系统变量下的"Classpath"如图，单击编辑，如没有，则单击"新建":

- "变量名"：ClassPath
- "变量值"：.;%SCALA_HOME%\bin;%SCALA_HOME%\lib\dt.jar;%SCALA_HOME%\lib\tools.jar.;

注意："变量值"最前面的 .; 不要漏掉。最后单击确定即可。



检查环境变量是否设置好了：调出"cmd"检查。单击 【开始】，在输入框中输入cmd，然后"回车"，输入 scala，然后回车，如环境变量设置ok，你应该能看到这些信息。



以下列出了不同系统放置的目录（可作为参考）：

系统环境	变量	值 (举例)
Unix	`$SCALA_HOME`	`/usr/local/share/scala`
	`$PATH`	`$PATH:$SCALA_HOME/bin`
Windows	`%SCALA_HOME%`	`c:\Progra~1\Scala`
	`%PATH%`	`%PATH%;%SCALA_HOME%\bin`

Scala 基础语法

如果你之前是一名 Java 程序员，并了解 Java 语言的基础知识，那么你能很快学会 Scala 的基础语法。

Scala 与 Java 的最大区别是：Scala 语句末尾的分号 ; 是可选的。

我们可以认为 Scala 程序是对象的集合，通过调用彼此的方法来实现消息传递。接下来我们来理解下，类，对象，方法，实例变量的概念：

- 对象 - 对象有属性和行为。例如：一只狗的状属性有：颜色，名字，行为有：叫、跑、吃等。对象是一个类的实例。
- 类 - 类是对象的抽象，而对象是类的具体实例。
- 方法 - 方法描述的基本的行为，一个类可以包含多个方法。
- 字段 - 每个对象都有它唯一的实例变量集合，即字段。对象的属性通过给字段赋值来创建。

第一个 Scala 程序

交互式编程

交互式编程不需要创建脚本文件，可以通过以下命令调用：

```
$ scala
Welcome to Scala version 2.11.7 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_31).
Type in expressions to have them evaluated.
Type :help for more information.

scala> 1 + 1
res0: Int = 2

scala> println("Hello World!")
Hello World!

scala>
```

脚本形式

我们也可以通过创建一个 HelloWorld.scala 的文件来执行代码，HelloWorld.scala 代码如下所示：

```
object HelloWorld {
   /* 这是我的第一个 Scala 程序
    * 以下程序将输出'Hello World!'
    */
   def main(args: Array[String]) {
      println("Hello, world!") // 输出 Hello World
   }
}
```

接下来我们使用 scalac 命令编译它：

```
$ scalac HelloWorld.scala
$ ls
HelloWorld$.class    HelloWorld.scala
HelloWorld.class
```

编译后我们可以看到目录下生成了 HelloWorld.class 文件，该文件可以在Java Virtual Machine (JVM)上运行。

编译后，我们可以使用以下命令来执行程序：

```
$ scala HelloWorld
Hello, world!
```

在线实例？

基本语法

Scala 基本语法需要注意以下几点：

- 区分大小写 - Scala是大小写敏感的，这意味着标识Hello 和 hello在Scala中会有不同的含义。
- 类名 - 对于所有的类名的第一个字母要大写。

  如果需要使用几个单词来构成一个类的名称，每个单词的第一个字母要大写。

  示例：*class MyFirstScalaClass*
- 方法名称 - 所有的方法名称的第一个字母用小写。

  如果若干单词被用于构成方法的名称，则每个单词的第一个字母应大写。

  示例：*def myMethodName()*
- 程序文件名 - 程序文件的名称应该与对象名称完全匹配。

  保存文件时，应该保存它使用的对象名称（记住Scala是区分大小写），并追加".scala"为文件扩展名。（如果文件名和对象名称不匹配，程序将无法编译）。

示例: 假设"HelloWorld"是对象的名称。那么该文件应保存为'HelloWorld.scala"

- **def main(args: Array[String])** - Scala程序从main()方法开始处理，这是每一个Scala程序的强制程序入口部分。

标识符

Scala 可以使用两种形式的标志符，字符数字和符号。

字符数字使用字母或是下划线开头，后面可以接字母或是数字，符号"$"在 Scala 中也看作为字母。然而以"$"开头的标识符为保留的 Scala 编译器产生的标志符使用，应用程序应该避免使用"$"开始的标识符，以免造成冲突。

Scala 的命名规则采用和 Java 类似的 camel 命名规则，首字符小写，比如 toString。类名的首字符还是使用大写。此外也应该避免使用以下划线结尾的标志符以避免冲突。符号标志符包含一个或多个符号，如+，:，? 等，比如:

```
+ ++ ::: < ?> :->
```

Scala 内部实现时会使用转义的标志符，比如:-> 使用 $colon$minus$greater 来表示这个符号。因此如果你需要在 Java 代码中访问:->方法，你需要使用 Scala 的内部名称 $colon$minus$greater。

混合标志符由字符数字标志符后面跟着一个或多个符号组成，比如 unary_+ 为 Scala 对+方法的内部实现时的名称。字面量标志符为使用"定义的字符串，比如 `x` `yield`。

你可以在"之间使用任何有效的 Scala 标志符，Scala 将它们解释为一个 Scala 标志符，一个典型的使用为 Thread 的 yield 方法， 在 Scala 中你不能使用 Thread.yield()是因为 yield 为 Scala 中的关键字， 你必须使用 Thread.`yield`()来使用这个方法。

Scala 关键字

下表列出了 scala 保留关键字，我们不能使用以下关键字作为变量：

abstract	case	catch	class
def	do	else	extends
false	final	finally	for
forSome	if	implicit	import
lazy	match	new	null
object	override	package	private
protected	return	sealed	super
this	throw	trait	try
true	type	val	var
while	with	yield	
-	:	=	=>
<-	<:	<%	>:
#	@		

Scala 注释

Scala 类似 Java 支持单行很多行注释。多行注释可以嵌套，但必须正确嵌套，一个注释开始符号对应一个结束符号。注释在 Scala 编译中会被忽略，实例如下：

```
object HelloWorld {
   /* 这是一个 Scala 程序
    * 这是一行注释
    * 这里演示了多行注释
    */
   def main(args: Array[String]) {
      // 输出 Hello World
      // 这是一个单行注释
      println("Hello, world!")
   }
}
```

空行和空格

一行中只有空格或者带有注释，Scala 会认为其是空行，会忽略它。标记可以被空格或者注释来分割。

换行符

Scala是面向行的语言，语句可以用分号（;）结束或换行符。Scala 程序里,语句末尾的分号通常是可选的。如果你愿意可以输入一个,但若一行里仅 有一个语句也可不写。 另一方面,如果一行里写多个语句那么分号是需要的。例如

```
val s = "菜鸟教程"; println(s)
```

Scala 包

定义包

Scala 使用 package 关键字定义包，在Scala将代码定义到某个包中有两种方式：

第一种方法和 Java 一样，在文件的头定义包名，这种方法就后续所有代码都放在该报中。比如：

```
package com.runoob
class HelloWorld
```

第二种方法有些类似 C#，如：

```
package com.runoob {
  class HelloWorld
}
```

第二种方法，可以在一个文件中定义多个包。

引用

Scala 使用 import 关键字引用包。

```
import java.awt.Color  // 引入Color

import java.awt._  // 引入包内所有成员

def handler(evt: event.ActionEvent) { // java.awt.event.ActionEvent
  ...  // 因为引入了java.awt，所以可以省去前面的部分
}
```

import语句可以出现在任何地方，而不是只能在文件顶部。import的效果从开始延伸到语句块的结束。这可以大幅减少名称冲突的可能性。

如果想要引入包中的几个成员，可以使用selector（选取器）：

```
import java.awt.{Color, Font}

// 重命名成员
import java.util.{HashMap => JavaHashMap}

// 隐藏成员
import java.util.{HashMap => _, _} // 引入了util包的所有成员，但是HashMap被隐藏了
```

> 注意：默认情况下，Scala 总会引入 java.lang. 、 *scala.* 和 Predef._，这里也能解释，为什么以scala开头的包，在使用时都是省去scala.的。

Scala 数据类型

Scala 与 Java有着相同的数据类型，下表列出了 Scala 支持的数据类型：

数据类型	描述
Byte	8位有符号补码整数。数值区间为 -128 到 127
Short	16位有符号补码整数。数值区间为 -32768 到 32767
Int	32位有符号补码整数。数值区间为 -2147483648 到 2147483647
Long	64位有符号补码整数。数值区间为 -9223372036854775808 到 9223372036854775807
Float	32位IEEE754单精度浮点数
Double	64位IEEE754单精度浮点数
Char	16位无符号Unicode字符, 区间值为 U+0000 到 U+FFFF
String	字符序列
Boolean	true或false
Unit	表示无值，和其他语言中void等同。用作不返回任何结果的方法的结果类型。Unit只有一个实例值，写成()。
Null	null 或空引用
Nothing	Nothing类型在Scala的类层级的最低端；它是任何其他类型的子类型。
Any	Any是所有其他类的超类
AnyRef	AnyRef类是Scala里所有引用类(reference class)的基类

上表中列出的数据类型都是对象，也就是说scala没有java中的原生类型。在scala是可以对数字等基础类型调用方法的。

Scala 基础字面量

Scala 非常简单且直观。接下来我们会详细介绍 Scala 字面量。

整型字面量

整型字面量用于 Int 类型，如果表示 Long，可以在数字后面添加 L 或者小写 l 作为后缀。：

```
0
035
21
0xFFFFFFFF
0777L
```

浮点型字面量

如果浮点数后面有f或者F后缀时，表示这是一个Float类型，否则就是一个Double类型的。实例如下：

```
0.0
1e30f
3.14159f
1.0e100
.1
```

布尔型字面量

布尔型字面量有 true 和 false。

符号字面量

符号字面量被写成： **'<标识符>**， 这里 **<标识符>** 可以是任何字母或数字的标识（注意：不能以数字开头）。这种字面量被映射成预定义类scala.Symbol的实例。

如： 符号字面量 **'x** 是表达式 **scala.Symbol("x")** 的简写，符号字面量定义如下：

```
package scala
final case class Symbol private (name: String) {
   override def toString: String = "'" + name
}
```

字符字面量

在scala中字符类型表示为半角单引号(')中的字符，如下：

```
'a'
'\u0041'
'\n'
'\t'
```

其中 \ 表示转移字符，其后可以跟 **u0041** 数字或者 **\r\n** 等固定的转义字符。

字符串字面量

字符串表示方法是在双引号中(") 包含一系列字符，如：

```
"Hello,\nWorld!"
"菜鸟教程官网：www.runoob.com"
```

多行字符串的表示方法

多行字符串用三个双引号来表示分隔符，格式为：**""" ... """**。

实例如下：

```
val foo = """菜鸟教程
www.runoob.com
www.w3cschool.cc
www.runnoob.com
以上三个地址都能访问"""
```

Null 值

空值是 scala.Null 类型。

Scala.Null和scala.Nothing是用统一的方式处理Scala面向对象类型系统的某些"边界情况"的特殊类型。

Null类是null引用对象的类型，它是每个引用类（继承自AnyRef的类）的子类。Null不兼容值类型。

Scala 转义字符

下表列出了常见的转义字符：

转义字符	Unicode	描述
\b	\u0008	退格(BS)，将当前位置移到前一列
\t	\u0009	水平制表(HT) （跳到下一个TAB位置）
\n	\u000c	换行(LF)，将当前位置移到下一行开头
\f	\u000c	换页(FF)，将当前位置移到下页开头
\r	\u000d	回车(CR)，将当前位置移到本行开头
\"	\u0022	代表一个双引号(")字符
\'	\u0027	代表一个单引号（'）字符
\\	\u005c	代表一个反斜线字符 '\'

0 到 255 间的 Unicode 字符可以用一个八进制转义序列来表示，即反斜线?\?后跟 最多三个八进制。

在字符或字符串中，反斜线和后面的字符序列不能构成一个合法的转义序列将会导致 编译错误。

以下实例演示了一些转义字符的使用：

```
object Test {
   def main(args: Array[String]) {
      println("Hello\tWorld\n\n" );
   }
}
```

运行实例？

执行以上代码输出结果如下所示：

```
$ scalac Test.scala
$ scala Test
Hello    World

$
```

Scala 变量

变量是一种使用方便的占位符，用于引用计算机内存地址，变量创建后会占用一定的内存空间。

基于变量的数据类型，操作系统会进行内存分配并且决定什么将被储存在保留内存中。因此，通过给变量分配不同的数据类型，你可以在这些变量中存储整数，小数或者字字母。

变量声明

在学习如何声明变量与常量之前，我们先来了解一些变量与常量。

- 一、变量：在程序运行过程中其值可能发生改变的量叫做变量。如：时间，年龄。
- 二、常量 在程序运行过程中其值不会发生变化的量叫做常量。如：数值 3，字符'A'。

在 Scala 中，使用关键词 **"var"** 声明变量，使用关键词 **"val"** 声明常量。

声明变量实例如下：

```
var myVar : String = "Foo"
var myVar : String = "Too"
```

以上定义了变量 myVar，我们可以修改它。

声明常量实例如下：

```
val myVal :  String  =  "Foo"
```

以上定义了常量 myVal，它是不能修改的。如果程序尝试修改常量 myVal 的值，程序将会在编译时报错。

变量类型声明

变量的类型在变量名之后等号之前声明。定义变量的类型的语法格式如下：

```
var VariableName : DataType [=  Initial Value]

或

val VariableName : DataType [=  Initial Value]
```

变量声明不一定需要初始值，以下也是正确的：

```
var myVar :Int;
val myVal :String;
```

变量类型引用

在 Scala 中声明变量和常量不一定要指明数据类型，在没有指明数据类型的情况下，其数据类型是通过变量或常量的初始值推断出来的。

所以，如果在没有指明数据类型的情况下声明变量或常量必须要给出其初始值，否则将会报错。

```
var myVar = 10;
val myVal = "Hello, Scala!";
```

以上实例中，myVar 会被推断为 Int 类型，myVal 会被推断为 String 类型。

Scala 多个变量声明

Scala 支持多个变量的声明：

```
val xmax, ymax = 100  // xmax, ymax都声明为100
```

如果方法返回值是元组，我们可以使用 val 来声明一个元组：

```
val (myVar1: Int, myVar2: String) = Pair(40, "Foo")
```

也可以不指定数据类型：

```
val (myVar1, myVar2) = Pair(40, "Foo")
```

Scala 访问修饰符

Scala 访问修饰符基本和Java的一样，分别有：private，protected，public。

如果没有指定访问修饰符符，默认情况下，Scala对象的访问级别都是 public。

Scala 中的 private 限定符，比 Java 更严格，在嵌套类情况下，外层类甚至不能访问被嵌套类的私有成员。

私有(Private)成员

用private关键字修饰，带有此标记的成员仅在包含了成员定义的类或对象内部可见，同样的规则还适用内部类。

```
class Outer{
    class Inner{
    private def f(){println("f")}
    class InnerMost{
        f() // 正确
        }
    }
    (new Inner).f() //错误
}
```

(new Inner).f( ) 访问不合法是因为 f 在 Inner 中被声明为 private，而访问不在类Inner之内。

但在 InnerMost 里访问f就没有问题的，因为这个访问包含在 Inner 类之内。

Java中允许这两种访问，因为它允许外部类访问内部类的私有成员。

保护(Protected)成员

在 scala 中，对保护（Protected）成员的访问比 java 更严格一些。因为它只允许保护成员在定义了该成员的的类的子类中被访问。而在java中，用protected关键字修饰的成员，除了定义了该成员的类的子类可以访问，同一个包里的其他类也可以进行访问。

```
package p{
class Super{
    protected def f() {println("f")}
    }
    class Sub extends Super{
        f()
    }
    class Other{
        (new Super).f() //错误
    }
}
```

上例中，Sub 类对 f 的访问没有问题，因为 f 在 Super 中被声明为 protected，而 Sub 是 Super 的子类。相反，Other 对 f 的访问不被允许，因为 other 没有继承自 Super。而后者在 java 里同样被认可，因为 Other 与 Sub 在同一包里。

公共(Public)成员

Scala中，如果没有指定任何的修饰符，则默认为 public。这样的成员在任何地方都可以被访问。

```
class Outer {
   class Inner {
      def f() { println("f") }
      class InnerMost {
         f() // 正确
      }
   }
   (new Inner).f() // 正确因为 f() 是 public
}
```

作用域保护

Scala中，访问修饰符可以通过使用限定词强调。格式为:

```
private[x]

或

protected[x]
```

这里的x指代某个所属的包、类或单例对象。如果写成private[x],读作"这个成员除了对[…]中的类或[…]中的包中的类及它们的伴生对像可见外，对其它所有类都是private。

这种技巧在横跨了若干包的大型项目中非常有用，它允许你定义一些在你项目的若干子包中可见但对于项目外部的客户却始终不可见的东西。

```
package bobsrocckets{
    package navigation{
        private[bobsrockets] class Navigator{
         protected[navigation] def useStarChart(){}
         class LegOfJourney{
             private[Navigator] val distance = 100
             }
            private[this] var speed = 200
            }
        }
        package launch{
        import navigation._
        object Vehicle{
        private[launch] val guide = new Navigator
        }
    }
}
```

上述例子中，类Navigator被标记为private[bobsrockets]就是说这个类对包含在bobsrockets包里的所有的类和对象可见。

比如说，从Vehicle对象里对Navigator的访问是被允许的，因为对象Vehicle包含在包launch中，而launch包在bobsrockets中，相反，所有在包bobsrockets之外的代码都不能访问类Navigator。

Scala 运算符

一个运算符是一个符号，用于告诉编译器来执行指定的数学运算和逻辑运算。

Scala 含有丰富的内置运算符，包括以下几种类型：

- 算术运算符
- 关系运算符
- 逻辑运算符
- 位运算符
- 赋值运算符

接下来我们将为大家详细介绍以上各种运算符的应用。

算术运算符

下表列出了 Scala 支持的算术运算符。

假定变量 A 为 10，B 为 20：

运算符	描述	实例
+	加号	A + B 运算结果为 30
-	减号	A - B 运算结果为 -10
*	乘号	A * B 运算结果为 200
/	除号	B / A 运算结果为 2
%	取余	B % A 运算结果为 0

实例

```
object Test {
   def main(args: Array[String]) {
      var a = 10;
      var b = 20;
      var c = 25;
      var d = 25;
      println("a + b = " + (a + b) );
      println("a - b = " + (a - b) );
      println("a * b = " + (a * b) );
      println("b / a = " + (b / a) );
      println("b % a = " + (b % a) );
      println("c % a = " + (c % a) );

   }
}
```

运行实例？

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
a + b = 30
a - b = -10
a * b = 200
b / a = 2
b % a = 0
c % a = 5
```

关系运算符

下表列出了 Scala 支持的关系运算符。

假定变量 A 为 10，B 为 20：

运算符	描述	实例
==	等于	(A == B) 运算结果为 false
!=	不等于	(A != B) 运算结果为 true
>	大于	(A > B) 运算结果为 false
<	小于	(A < B) 运算结果为 true
>=	大于等于	(A >= B) 运算结果为 false
<=	小于等于	(A <= B) 运算结果为 true

实例

```
object Test {
   def main(args: Array[String]) {
      var a = 10;
      var b = 20;
      println("a == b = " + (a == b) );
      println("a != b = " + (a != b) );
      println("a > b = " + (a > b) );
      println("a < b = " + (a < b) );
      println("b >= a = " + (b >= a) );
      println("b <= a = " + (b <= a) );
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
a == b = false
a != b = true
a > b = false
a < b = true
b >= a = true
b <= a = false
```

逻辑运算符

下表列出了 Scala 支持的逻辑运算符。

假定变量 A 为 1，B 为 0：

运算符	描述	实例
&&	逻辑与	(A && B) 运算结果为 false
||	逻辑或	(A || B) 运算结果为 true
!	逻辑非	!(A && B) 运算结果为 true

实例

```
object Test {
   def main(args: Array[String]) {
      var a = true;
      var b = false;

      println("a && b = " + (a&&b) );

      println("a || b = " + (a||b) );

      println("!(a && b) = " + !(a && b) );
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
a && b = false
a || b = true
!(a && b) = true
```

位运算符

位运算符用来对二进制位进行操作，~,&,|,^分别为取反，按位与与，按位与或，按位与异或运算，如下表实例：

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

如果指定 A = 60; 及 B = 13; 两个变量对应的二进制为：

```
A = 0011 1100

B = 0000 1101

-------位运算----------

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A  = 1100 0011
```

Scala 中的按位运算法则如下：

运算符	描述	实例
&	按位与运算符	(a & b) 输出结果 12 ，二进制解释： 0000 1100
|	按位或运算符	(a | b) 输出结果 61 ，二进制解释： 0011 1101
^	按位异或运算符	(a ^ b) 输出结果 49 ，二进制解释： 0011 0001
~	按位取反运算符	(~a) 输出结果 -61 ，二进制解释： 1100 0011， 在一个有符号二进制数的补码形式。
<<	左移动运算符	a << 2 输出结果 240 ，二进制解释： 1111 0000
>>	右移动运算符	a >> 2 输出结果 15 ，二进制解释： 0000 1111
>>>	无符号右移	A >>>2 输出结果 15, 二进制解释: 0000 1111

实例

```
object Test {
   def main(args: Array[String]) {
      var a = 60;           /* 60 = 0011 1100 */
      var b = 13;           /* 13 = 0000 1101 */
      var c = 0;

      c = a & b;            /* 12 = 0000 1100 */
      println("a & b = " + c );

      c = a | b;            /* 61 = 0011 1101 */
      println("a | b = " + c );

      c = a ^ b;            /* 49 = 0011 0001 */
      println("a ^ b = " + c );

      c = ~a;               /* -61 = 1100 0011 */
      println("~a = " + c );

      c = a << 2;           /* 240 = 1111 0000 */
      println("a << 2 = " + c );

      c = a >> 2;           /* 215 = 1111 */
      println("a >> 2  = " + c );

      c = a >>> 2;          /* 215 = 0000 1111 */
      println("a >>> 2 = " + c );
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
a & b = 12
a | b = 61
a ^ b = 49
~a = -61
a << 2 = 240
a >> 2  = 15
a >>> 2 = 15
```

赋值运算符

以下列出了 Scala 语言支持的赋值运算符:

运算符	描述	实例
=	简单的赋值运算，指定右边操作数赋值给左边的操作数。	C = A + B 将 A + B 的运算结果赋值给 C
+=	相加后再赋值，将左右两边的操作数相加后再赋值给左边的操作数。	C += A 相当于 C = C + A
-=	相减后再赋值，将左右两边的操作数相减后再赋值给左边的操作数。	C -= A 相当于 C = C - A
*=	相乘后再赋值，将左右两边的操作数相乘后再赋值给左边的操作数。	C = *A* 相当于 *C* = *C* A
/=	相除后再赋值，将左右两边的操作数相除后再赋值给左边的操作数。	C /= A 相当于 C = C / A
%=	求余后再赋值，将左右两边的操作数求余后再赋值给左边的操作数。	C %= A is equivalent to C = C % A
<<=	按位左移后再赋值	C <<= 2 相当于 C = C << 2
>>=	按位右移后再赋值	C >>= 2 相当于 C = C >> 2
&=	按位与运算后赋值	C &= 2 相当于 C = C & 2
^=	按位异或运算符后再赋值	C ^= 2 相当于 C = C ^ 2
|=	按位或运算后再赋值	C |= 2 相当于 C = C | 2

实例

```
object Test {
   def main(args: Array[String]) {
      var a = 10;
      var b = 20;
      var c = 0;

      c = a + b;
      println("c = a + b  = " + c );

      c += a ;
      println("c += a  = " + c );

      c -= a ;
      println("c -= a = " + c );

      c *= a ;
      println("c *= a = " + c );

      a = 10;
      c = 15;
      c /= a ;
      println("c /= a  = " + c );

      a = 10;
      c = 15;
      c %= a ;
      println("c %= a  = " + c );

      c <<= 2 ;
      println("c <<= 2  = " + c );

      c >>= 2 ;
      println("c >>= 2  = " + c );

      c >>= 2 ;
      println("c >>= a  = " + c );

      c &= a ;
      println("c &= 2  = " + c );

      c ^= a ;
      println("c ^= a  = " + c );

      c |= a ;
      println("c |= a  = " + c );
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
c = a + b  = 30
c += a  = 40
c -= a = 30
c *= a = 300
c /= a  = 1
c %= a  = 5
c <<= 2  = 20
c >>= 2  = 5
c >>= a  = 1
c &= 2  = 0
c ^= a  = 10
c |= a  = 10
```

运算符优先级

在一个表达式中可能包含多个有不同运算符连接起来的、具有不同数据类型的数据对象；由于表达式有多种运算，不同的运算顺序可能得出不同结果甚至出现错误运算错误，因为当表达式中含多种运算时，必须按一定顺序进行结合，才能保证运算的合理性和结果的正确性、唯一性。

优先级从上到下依次递减，最上面具有最高的优先级，逗号操作符具有最低的优先级。

相同优先级中，按结合顺序计算。大多数运算是从左至右计算，只有三个优先级是从右至左结合的，它们是单目运算符、条件运算符、赋值运算符。

基本的优先级需要记住：

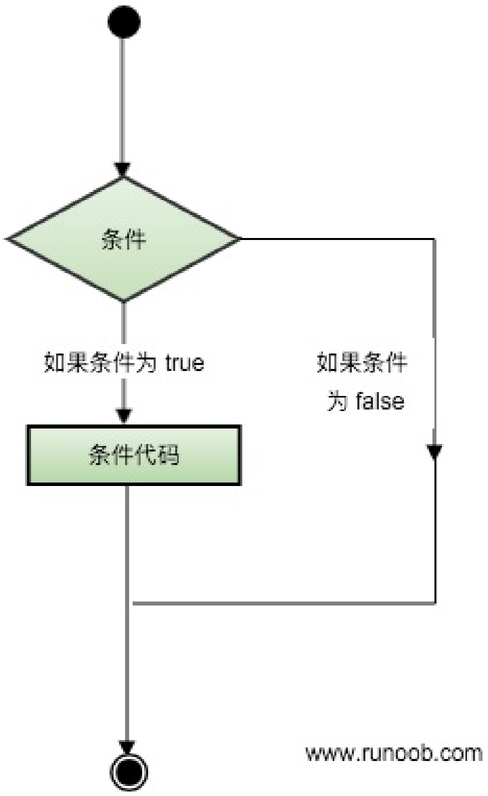
- 指针最优，单目运算优于双目运算。如正负号。
- 先乘除（模），后加减。
- 先算术运算，后移位运算，最后位运算。请特别注意：1 << 3 + 2 & 7 等价于 (1 << (3 + 2))&7
- 逻辑运算最后计算

运算符类型	运算符	结合方向
表达式运算	() [] . expr++ expr--	左到右
一元运算符	& + - ! ~ ++expr --expr * / % + - >> << > < >= <= == !=	右到左
位运算符	& ^ | && ||	左到右
三元运算符	?:	右到左
赋值运算符	= += -= *= /= %= >>= <<= &= ^= |=	右到左
逗号	,	左到右

Scala IF...ELSE 语句

Scala IF...ELSE 语句是通过一条或多条语句的执行结果（True或者False）来决定执行的代码块。

可以通过下图来简单了解条件语句的执行过程:



if 语句

if 语句有布尔表达式及之后的语句块组成。

语法

if 语句的语法格式如下：

```
if(布尔表达式)
{
   // 如果布尔表达式为 true 则执行该语句块
}
```

如果布尔表达式为 true 则执行大括号内的语句块，否则跳过大括号内的语句块，执行大括号之后的语句块。

实例

```
object Test {
   def main(args: Array[String]) {
      var x = 10;

      if( x < 20 ){
         println("x < 20");
      }
   }
}
```

运行实例？

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
x < 20
```

if...else 语句

if 语句后可以紧跟 else 语句，else 内的语句块可以在布尔表达式为 false 的时候执行。

语法

if...else 的语法格式如下：

```
if(布尔表达式){
   // 如果布尔表达式为 true 则执行该语句块
}else{
   // 如果布尔表达式为 false 则执行该语句块
}
```

实例

```
object Test {
   def main(args: Array[String]) {
      var x = 30;

      if( x < 20 ){
         println("x 小于 20");
      }else{
         println("x 大于 20");
      }
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
x 大于 20
```

if...else if...else 语句

if 语句后可以紧跟 else if...else 语句，在多个条件判断语句的情况下很有用。

语法

if...else if...else 语法格式如下：

```
if(布尔表达式 1){
   // 如果布尔表达式 1 为 true 则执行该语句块
}else if(布尔表达式 2){
   // 如果布尔表达式 2 为 true 则执行该语句块
}else if(布尔表达式 3){
   // 如果布尔表达式 3 为 true 则执行该语句块
}else {
   // 如果以上条件都为 false 执行该语句块
}
```

实例

```
object Test {
   def main(args: Array[String]) {
      var x = 30;

      if( x == 10 ){
         println("X 的值为 10");
      }else if( x == 20 ){
         println("X 的值为 20");
      }else if( x == 30 ){
         println("X 的值为 30");
      }else{
         println("无法判断 X 的值");
      }
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
X 的值为 30
```

if...else 嵌套语句

if...else 嵌套语句可以实现在 if 语句内嵌入一个或多个 if 语句。

语法

if...else 嵌套语句语法格式如下：

```
if(布尔表达式 1){
   // 如果布尔表达式 1 为 true 则执行该语句块
   if(布尔表达式 2){
      // 如果布尔表达式 2 为 true 则执行该语句块
   }
}
```

else if...else 的嵌套语句 类似 if...else 嵌套语句。

实例

```
object Test {
   def main(args: Array[String]) {
        var x = 30;
        var y = 10;

         if( x == 30 ){
            if( y == 10 ){
            println("X = 30 , Y = 10");
         }
      }
   }
}
```

执行以上代码，输出结果为：

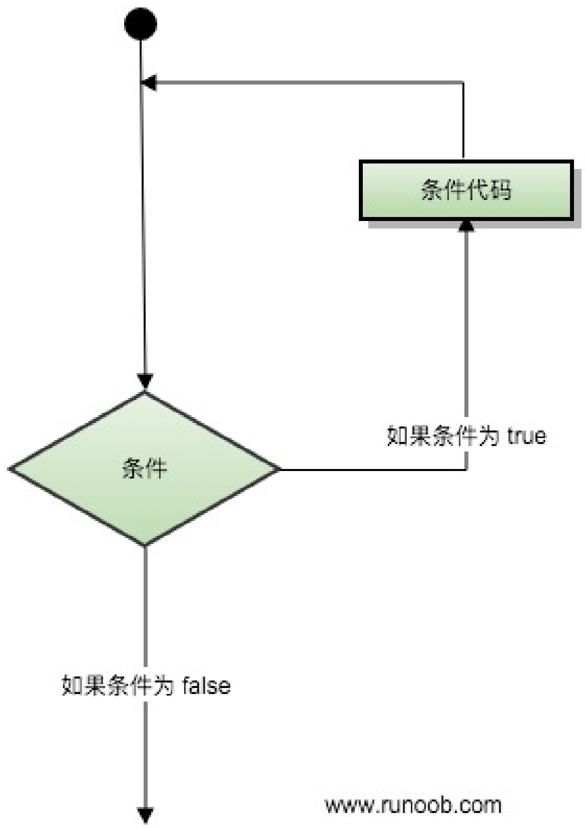
```
$ scalac Test.scala
$ scala Test
X =  30  , Y =  10
```

Scala 循环

有的时候，我们可能需要多次执行同一块代码。一般情况下，语句是按顺序执行的：函数中的第一个语句先执行，接着是第二个语句，依此类推。

编程语言提供了更为复杂执行路径的多种控制结构。

循环语句允许我们多次执行一个语句或语句组，下面是大多数编程语言中循环语句的流程图：



循环类型

Scala 语言提供了以下几种循环类型。点击链接查看每个类型的细节。

循环类型	描述
while 循环	运行一系列语句，如果条件为true，会重复运行，直到条件变为false。
do...while 循环	类似 while 语句区别在于判断循环条件之前，先执行一次循环的代码块。
for 循环	用来重复执行一系列语句直到达成特定条件达成，一般通过在每次循环完成后增加计数器的值来实现。

循环控制语句

循环控制语句改变你代码的执行顺序，通过它你可以实现代码的跳转。Scala 以下几种循环控制语句：

Scala 不支持 break 或 continue 语句，但从 2.8 版本后提供了一种中断循环的方式，点击以下链接查看详情。

控制语句	描述
break 语句	中断循环

无限循环

如果条件永远为 true，则循环将变成无限循环。我们可以使用 while 语句来实现无限循环：

```
object Test {
   def main(args: Array[String]) {
      var a = 10;
      // 无限循环
      while( true ){
         println( "a 的值为 : " + a );
      }
   }
}
```

以上代码执行后循环会永久执行下去，你可以使用 Ctrl + C 键来中断无限循环。

Scala while 循环

只要给定的条件为 true，Scala 语言中的 **while** 循环语句会重复执行循环体内的代码块。

语法

Scala 语言中 **while** 循环的语法：

```
while(condition)
{
   statement(s);
}
```

在这里，**statement(s)** 可以是一个单独的语句，也可以是几个语句组成的代码块。**condition** 可以是任意的表达式，当为任意非零值时都为 true。当条件为 true 时执行循环。

当条件为 false 时，程序流将继续执行紧接着循环的下一条语句。

流程图

在这里，*while* 循环的关键点是循环可能一次都不会执行。当条件为 false 时，会跳过循环主体，直接执行紧接着 while 循环的下一条语句。

实例

```
object Test {
   def main(args: Array[String]) {
      // 局部变量
      var a = 10;

      // while 循环执行
      while( a < 20 ){
         println( "Value of a: " + a );
         a = a + 1;
      }
   }
}
```

执行以上代码输出结果为：

```
$ scalac Test.scala
$ scala Test
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Scala do...while 循环

不像 while 循环在循环头部测试循环条件, Scala 语言中，do...while 循环是在循环的尾部检查它的条件。

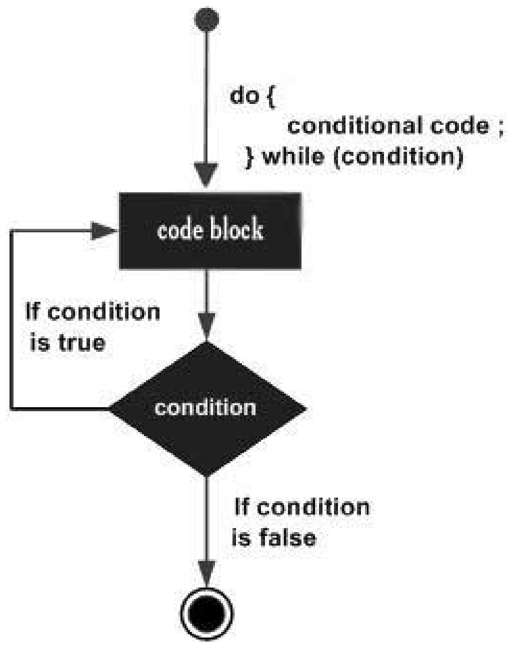
do...while 循环与 while 循环类似，但是 do...while 循环会确保至少执行一次循环。

语法

Scala 语言中 **while** 循环的语法：

```
do {
   statement(s);
} while( condition );
```

流程图



请注意，条件表达式出现在循环的尾部，所以循环中的 statement(s) 会在条件被测试之前至少执行一次。

如果条件为 true，控制流会跳转回上面的 do，然后重新执行循环中的 statement(s)。

这个过程会不断重复，直到给定条件变为 false 为止。

实例

```
object Test {
   def main(args: Array[String]) {
      // 局部变量
      var a = 10;

      // do 循环
      do{
         println( "Value of a: " + a );
         a = a + 1;
      }while( a < 20 )
   }
}
```

执行以上代码输出结果为：

```
$ scalac Test.scala
$ scala Test
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Scala do...while 循环

for 循环允许您编写一个执行指定次数的循环控制结构。

语法

Scala 语言中 **for** 循环的语法：

```
for( var x <- Range ){
   statement(s);
}
```

以上语法中，**Range** 可以是一个数字区间表示 **i to j** ，或者 **i until j**。左箭头 **<-** 用于为变量 x 赋值。

实例

以下是一个使用了 **i to j** 语法(包含 j)的实例:

```
object Test {
   def main(args: Array[String]) {
      var a = 0;
      // for 循环
      for( a <- 1 to 10){
         println( "Value of a: " + a );
      }
   }
}
```

执行以上代码输出结果为：

```
$ scalac Test.scala
$ scala Test
value of a: 1
value of a: 2
value of a: 3
value of a: 4
value of a: 5
value of a: 6
value of a: 7
value of a: 8
value of a: 9
value of a: 10
```

以下是一个使用了 **i until j** 语法(不包含 j)的实例:

```
object Test {
   def main(args: Array[String]) {
      var a = 0;
      // for 循环
      for( a <- 1 until 10){
         println( "Value of a: " + a );
      }
   }
}
```

执行以上代码输出结果为：

```
$ scalac Test.scala
$ scala Test
value of a: 1
value of a: 2
value of a: 3
value of a: 4
value of a: 5
value of a: 6
value of a: 7
value of a: 8
value of a: 9
```

在 **for** 循环 中你可以使用分号 (;) 来设置多个区间，它将迭代给定区间所有的可能值。以下实例演示了两个区间的循环实例：

```
object Test {
   def main(args: Array[String]) {
      var a = 0;
      var b = 0;
      // for 循环
      for( a <- 1 to 3; b <- 1 to 3){
         println( "Value of a: " + a );
         println( "Value of b: " + b );
      }
   }
}
```

执行以上代码输出结果为：

```
$ scalac Test.scala
$ scala Test
Value of a: 1
Value of b: 1
Value of a: 1
Value of b: 2
Value of a: 1
Value of b: 3
Value of a: 2
Value of b: 1
Value of a: 2
Value of b: 2
Value of a: 2
Value of b: 3
Value of a: 3
Value of b: 1
Value of a: 3
Value of b: 2
Value of a: 3
Value of b: 3
```

for 循环集合

for 循环集合的语法如下：

```
for( var x <- List ){
   statement(s);
}
```

以上语法中， **List** 变量是一个集合，for 循环会迭代所有集合的元素。

实例

以下实例将循环数字集合。我们使用 *List()* 来创建集合。再以后章节我们会详细介绍集合。

```
object Test {
   def main(args: Array[String]) {
      var a = 0;
      val numList = List(1,2,3,4,5,6);

      // for 循环
      for( a <- numList ){
         println( "Value of a: " + a );
      }
   }
}
```

执行以上代码输出结果为：

```
$ scalac Test.scala
$ scala Test
value of a: 1
value of a: 2
value of a: 3
value of a: 4
value of a: 5
value of a: 6
```

for 循环过滤

Scala 可以使用一个或多个 **if** 语句来过滤一些元素。

以下是在 for 循环中使用过滤器的语法。

```
for( var x <- List
      if condition1; if condition2...
   ){
   statement(s);
```

你可以使用分号(;)来为表达式添加一个或多个的过滤条件。

实例

以下是 for 循环中过滤的实例：

```
object Test {
   def main(args: Array[String]) {
      var a = 0;
      val numList = List(1,2,3,4,5,6,7,8,9,10);

      // for 循环
      for( a <- numList
           if a != 3; if a < 8 ){
         println( "Value of a: " + a );
      }
   }
}
```

执行以上代码输出结果为：

```
$ scalac Test.scala
$ scala Test
value of a: 1
value of a: 2
value of a: 4
value of a: 5
value of a: 6
value of a: 7
```

for 使用 yield

你可以将 for 循环的返回值作为一个变量存储。语法格式如下：

```
var retVal = for{ var x <- List
     if condition1; if condition2...
}yield x
```

注意大括号中用于保存变量和条件，*retVal* 是变量， 循环中的 yield 会把当前的元素记下来，保存在集合中，循环结束后将返回该集合。

实例

以下实例演示了 for 循环中使用 yield：

```
object Test {
   def main(args: Array[String]) {
      var a = 0;
      val numList = List(1,2,3,4,5,6,7,8,9,10);

      // for 循环
      var retVal = for{ a <- numList
                        if a != 3; if a < 8
                      }yield a

      // 输出返回值
      for( a <- retVal){
         println( "Value of a: " + a );
      }
   }
}
```

执行以上代码输出结果为：

```
$ scalac Test.scala
$ scala Test
value of a: 1
value of a: 2
value of a: 4
value of a: 5
value of a: 6
value of a: 7
```

Scala break 语句

Scala 语言中默认是没有 break 语句，但是你在 Scala 2.8 版本后可以使用另外一种方式来实现 *break* 语句。当在循环中使用 **break** 语句，在执行到该语句时，就会中断循环并执行循环体之后的代码块。

语法

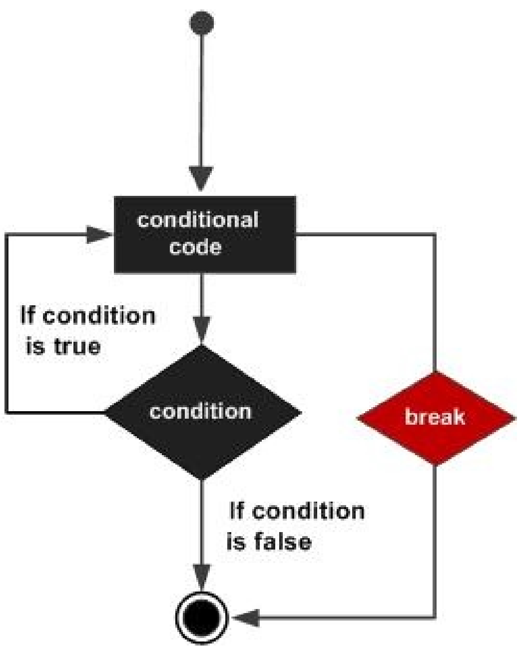
Scala 中 break 的语法有点不大一样，格式如下：

```
// 导入以下包
import scala.util.control._

// 创建 Breaks 对象
val loop = new Breaks;

// 在 breakable 中循环
loop.breakable{
    // 循环
    for(...){
       ....
       // 循环中断
       loop.break;
   }
}
```

流程图



实例

```
import scala.util.control._

object Test {
   def main(args: Array[String]) {
      var a = 0;
      val numList = List(1,2,3,4,5,6,7,8,9,10);

      val loop = new Breaks;
      loop.breakable {
         for( a <- numList){
            println( "Value of a: " + a );
            if( a == 4 ){
               loop.break;
            }
         }
      }
      println( "After the loop" );
   }
}
```

执行以上代码输出结果为：

```
$ scalac Test.scala
$ scala Test
Value of a: 1
Value of a: 2
Value of a: 3
Value of a: 4
After the loop
```

中断嵌套循环

以下实例演示了如何中断嵌套循环：

```
import scala.util.control._

object Test {
   def main(args: Array[String]) {
      var a = 0;
      var b = 0;
      val numList1 = List(1,2,3,4,5);
      val numList2 = List(11,12,13);

      val outer = new Breaks;
      val inner = new Breaks;

      outer.breakable {
         for( a <- numList1){
            println( "Value of a: " + a );
            inner.breakable {
               for( b <- numList2){
                  println( "Value of b: " + b );
                  if( b == 12 ){
                     inner.break;
                  }
               }
            } // 内嵌循环中断
         }
      } // 外部循环中断
   }
}
```

执行以上代码输出结果为：

```
$ scalac Test.scala
$ scala Test
Value of a: 1
Value of b: 11
Value of b: 12
Value of a: 2
Value of b: 11
Value of b: 12
Value of a: 3
Value of b: 11
Value of b: 12
Value of a: 4
Value of b: 11
Value of b: 12
Value of a: 5
Value of b: 11
Value of b: 12
```

Scala 函数

函数是一组一起执行一个任务的语句。 您可以把代码划分到不同的函数中。如何划分代码到不同的函数中是由您来决定的，但在逻辑上，划分通常是根据每个函数执行一个特定的任务来进行的。

Scala 有函数和方法，二者在语义上的区别很小。Scala 方法是类的一部分，而函数是一个对象可以赋值给一个变量。换句话来说在类中定义的函数即是方法。

我们可以在任何地方定义函数，甚至可以在函数内定义函数（内嵌函数）。更重要的一点是 Scala 函数名可以由以下特殊字符：**+, ++, ~, &,-, --** , **\\, /,** : 等。

函数声明

Scala 函数声明格式如下：

```
def functionName ([参数列表]) : [return type]
```

如果你不写等于号和方法主体，那么方法会被隐式声明为"抽象(abstract)"，包含它的类型于是也是一个抽象类型。

函数定义

方法定义由一个def 关键字开始，紧接着是可选的参数列表，一个冒号"：" 和方法的返回类型，一个等于号"="，最后是方法的主体。

Scala 函数定义格式如下：

```
def functionName ([参数列表]) : [return type] = {
   function body
   return [expr]
}
```

以上代码中 **return type** 可以是任意合法的 Scala 数据类型。参数列表中的参数可以使用逗号分隔。

以下函数的功能是将两个传入的参数相加并求和：

```
object add{
   def addInt( a:Int, b:Int ) : Int = {
      var sum:Int = 0
      sum = a + b

      return sum
   }
}
```

如果函数没有返回值，可以返回为 **Unit**，这个类似于 Java 的 **void**, 实例如下：

```
object Hello{
   def printMe( ) : Unit = {
      println("Hello, Scala!")
   }
}
```

函数调用

Scala 提供了多种不同的函数调用方式：

以下是调用方法的标准格式：

```
functionName( 参数列表 )
```

如果函数使用了实例的对象来调用，我们可以使用类似java的格式 (使用 **.** 号)：

```
[instance.]functionName( 参数列表 )
```

以上实例演示了定义与调用函数的实例:

```
object Test {
   def main(args: Array[String]) {
        println( "Returned Value : " + addInt(5,7) );
   }
   def addInt( a:Int, b:Int ) : Int = {
      var sum:Int = 0
      sum = a + b

      return sum
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
Returned Value : 12
```

Scala也是一种函数式语言，所以函数是 Scala 语言的核心。以下一些函数概念有助于我们更好的理解 Scala 编程：

函数概念解析接案例	
函数传名调用(Call-by-Name)	指定函数参数名
函数 - 可变参数	递归函数
默认参数值	高阶函数
内嵌函数	匿名函数
偏应用函数	函数柯里化(Function Currying)

Scala 函数传名调用(call-by-name)

Scala的解释器在解析函数参数(function arguments)时有两种方式：

- 传值调用（call-by-value）：先计算参数表达式的值，再应用到函数内部；
- 传名调用（call-by-name）：将未计算的参数表达式直接应用到函数内部

在进入函数内部前，传值调用方式就已经将参数表达式的值计算完毕，而传名调用是在函数内部进行参数表达式的值计算的。

这就造成了一种现象，每次使用传名调用时，解释器都会计算一次表达式的值。

```
object Test {
   def main(args: Array[String]) {
        delayed(time());
   }

   def time() = {
      println("获取时间，单位为纳秒")
      System.nanoTime
   }
   def delayed( t: => Long ) = {
      println("在 delayed 方法内")
      println("参数： " + t)
      t
   }
}
```

以上实例中我们声明了 delayed 方法， 该方法在变量名和变量类型使用 => 符号来设置传名调用。执行以上代码，输出结果如下：

```
$ scalac Test.scala
$ scala Test
在 delayed 方法内
获取时间，单位为纳秒
参数： 241550840475831
获取时间，单位为纳秒
```

实例中 delay 方法打印了一条信息表示进入了该方法，接着 delay 方法打印接收到的值，最后再返回 t。

Scala 指定函数参数名

一般情况下函数调用参数，就按照函数定义时的参数顺序一个个传递。但是我们也可以通过指定函数参数名，并且不需要按照顺序向函数传递参数，实例如下：

```
object Test {
   def main(args: Array[String]) {
        printInt(b=5, a=7);
   }
   def printInt( a:Int, b:Int ) = {
      println("Value of a : " + a );
      println("Value of b : " + b );
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
Value of a :  7
Value of b :  5
```

Scala 函数 - 可变参数

Scala 允许你指明函数的最后一个参数可以是重复的，即我们不需要指定函数参数的个数，可以向函数传入可变长度参数列表。

Scala 通过在参数的类型之后放一个星号来设置可变参数(可重复的参数)。例如：

```
object Test {
   def main(args: Array[String]) {
        printStrings("Runoob", "Scala", "Python");
   }
   def printStrings( args:String* ) = {
      var i : Int = 0;
      for( arg <- args ){
         println("Arg value[" + i + "] = " + arg );
         i = i + 1;
      }
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
Arg value[0] = Runoob
Arg value[1] = Scala
Arg value[2] = Python
```

Scala 递归函数

递归函数在函数式编程的语言中起着重要的作用。

Scala 同样支持递归函数。

递归函数意味着函数可以调用它本身。

以上实例使用递归函数来计算阶乘：

```
object Test {
   def main(args: Array[String]) {
      for (i <- 1 to 10)
         println(i + " 的阶乘为: = " + factorial(i) )
   }

   def factorial(n: BigInt): BigInt = {
      if (n <= 1)
         1
      else
      n * factorial(n - 1)
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
1 的阶乘为: = 1
2 的阶乘为: = 2
3 的阶乘为: = 6
4 的阶乘为: = 24
5 的阶乘为: = 120
6 的阶乘为: = 720
7 的阶乘为: = 5040
8 的阶乘为: = 40320
9 的阶乘为: = 362880
10 的阶乘为: = 3628800
```

Scala 高阶函数

高阶函数（Higher-Order Function）就是操作其他函数的函数。

Scala 中允许使用高阶函数, 高阶函数可以使用其他函数作为参数，或者使用函数作为输出结果。

以下实例中，apply() 函数使用了另外一个函数 f 和 值 v 作为参数，而函数 f 又调用了参数 v：

```
object Test {
   def main(args: Array[String]) {

      println( apply( layout, 10) )

   }
   // 函数 f 和 值 v 作为参数，而函数 f 又调用了参数 v
   def apply(f: Int => String, v: Int) = f(v)

   def layout[A](x: A) = "[" + x.toString() + "]"

}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
[10]
```

Scala 函数嵌套

我么可以在 Scala 函数内定义函数，定义在函数内的函数称之为局部函数。

以下实例我们实现阶乘运算，并使用内嵌函数：

```
object Test {
   def main(args: Array[String]) {
      println( factorial(0) )
      println( factorial(1) )
      println( factorial(2) )
      println( factorial(3) )
   }

   def factorial(i: Int): Int = {
      def fact(i: Int, accumulator: Int): Int = {
         if (i <= 1)
            accumulator
         else
            fact(i - 1, i * accumulator)
      }
      fact(i, 1)
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
1
1
2
6
```

Scala 匿名函数

Scala 中定义匿名函数的语法很简单，箭头左边是参数列表，右边是函数体，参数的类型是可省略的，Scala 的类型推测系统会推测出参数的类型。使用匿名函数后，我们的代码变得更简洁了。

下面的表达式就定义了一个接受一个Int类型输入参数的匿名函数:

```
var inc = (x:Int) => x+1
```

上述定义的匿名函数，其实是下面这种写法的简写：

```
def add2 = new Function1[Int,Int]{
    def apply(x:Int):Int = x+1;
}
```

以上实例的 inc 现在可作为一个函数，使用方式如下：

```
var x = inc(7)-1
```

同样我们可以在匿名函数中定义多个参数：

```
var mul = (x: Int, y: Int) => x*y
```

mul 现在可作为一个函数，使用方式如下：

```
println(mul(3, 4))
```

我们也可以不给匿名函数设置参数，如下所示：

```
var userDir = () => { System.getProperty("user.dir") }
```

userDir 现在可作为一个函数，使用方式如下：

```
println( userDir )
```

Scala 偏应用函数

Scala 偏应用函数是一种表达式，你不需要提供函数需要的所有参数，只需要提供部分，或不提供所需参数。

如下实例，我们打印日志信息：

```
import java.util.Date

object Test {
   def main(args: Array[String]) {
      val date = new Date
      log(date, "message1" )
      Thread.sleep(1000)
      log(date, "message2" )
      Thread.sleep(1000)
      log(date, "message3" )
   }

   def log(date: Date, message: String)  = {
     println(date + "----" + message)
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
Mon Dec 02 12:52:41 CST 2013----message1
Mon Dec 02 12:52:41 CST 2013----message2
Mon Dec 02 12:52:41 CST 2013----message3
```

实例中，log() 方法接收两个参数：date 和 message。我们在程序执行时调用了三次，参数 date 值都相同，message 不同。

我们可以使用偏应用函数优化以上方法，绑定第一个 date 参数，第二个参数使用下划线(_)替换缺失的参数列表，并把这个新的函数值的索引的赋给变量。以上实例修改如下：

```
import java.util.Date

object Test {
   def main(args: Array[String]) {
      val date = new Date
      val logWithDateBound = log(date, _ : String)

      logWithDateBound("message1" )
      Thread.sleep(1000)
      logWithDateBound("message2" )
      Thread.sleep(1000)
      logWithDateBound("message3" )
   }

   def log(date: Date, message: String)  = {
     println(date + "----" + message)
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
Mon Dec 02 12:53:56 CST 2013----message1
Mon Dec 02 12:53:56 CST 2013----message2
Mon Dec 02 12:53:56 CST 2013----message3
```

Scala 函数柯里化(Currying)

柯里化(Currying)指的是将原来接受两个参数的函数变成新的接受一个参数的函数的过程。新的函数返回一个以原有第二个参数为参数的函数。

实例

首先我们定义一个函数:

```
def add(x:Int,y:Int)=x+y
```

那么我们应用的时候，应该是这样用：add(1,2)

现在我们把这个函数变一下形：

```
def add(x:Int)(y:Int)  = x + y
```

那么我们应用的时候，应该是这样用：add(1)(2),最后结果都一样是3，这种方式（过程）就叫柯里化。

实现过程

add(1)(2) 实际上是依次调用两个普通函数（非柯里化函数），第一次调用使用一个参数 x，返回一个函数类型的值，第二次使用参数y调用这个函数类型的值。

实质上最先演变成这样一个方法：

```
def add(x:Int)=(y:Int)=>x+y
```

那么这个函数是什么意思呢？ 接收一个x为参数，返回一个匿名函数，该匿名函数的定义是：接收一个Int型参数y，函数体为x+y。现在我们来对这个方法进行调用。

```
val result = add(1)
```

返回一个result，那result的值应该是一个匿名函数：(y:Int)=>1+y

所以为了得到结果，我们继续调用result。

```
val sum = result(2)
```

最后打印出来的结果就是3。

完整实例

下面是一个完整实例：

```
object Test {
   def main(args: Array[String]) {
      val str1:String = "Hello, "
      val str2:String = "Scala!"
      println( "str1 + str2 = " +  strcat(str1)(str2) )
   }

   def strcat(s1: String)(s2: String) = {
      s1 + s2
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
str1 + str2 = Hello, Scala!
```

Scala 闭包

闭包是一个函数，返回值依赖于声明在函数外部的一个或多个变量。

闭包通常来讲可以简单的认为是可以访问一个函数里面局部变量的另外一个函数。

如下面这段匿名的函数：

```
val multiplier = (i:Int) => i * 10
```

函数体内有一个变量 i，它作为函数的一个参数。如下面的另一段代码：

```
val multiplier = (i:Int) => i * factor
```

在 multiplier 中有两个变量：i 和 factor。其中的一个 i 是函数的形式参数，在 multiplier 函数被调用时，i 被赋予一个新的值。然而，factor不是形式参数，而是自由变量，考虑下面代码：

```
var factor = 3
val multiplier = (i:Int) => i * factor
```

这里我们引入一个自由变量 factor，这个变量定义在函数外面。

这样定义的函数变量 multiplier 成为一个"闭包"，因为它引用到函数外面定义的变量，定义这个函数的过程是将这个自由变量捕获而构成一个封闭的函数。

完整实例

```
object Test {
   def main(args: Array[String]) {
      println( "muliplier(1) value = " +  multiplier(1) )
      println( "muliplier(2) value = " +  multiplier(2) )
   }
   var factor = 3
   val multiplier = (i:Int) => i * factor
}
```

运行实例 ?

执行以上代码，输出结果为：

```
$ scalac Test.scala
$  scala Test
muliplier(1) value = 3
muliplier(2) value = 6
```

Scala 字符串

以下实例将字符串赋值给一个常量：

```
object Test {
   val greeting: String = "Hello,World!"

   def main(args: Array[String]) {
      println( greeting )
   }
}
```

以上实例定义了变量 greeting，为字符串常量，它的类型为 String (java.lang.String)。

在 Scala 中，字符串的类型实际上是 Java String，它本身没有 String 类。

在 Scala 中，String 是一个不可变的对象，所以该对象不可被修改。这就意味着你如果修改字符串就会产生一个新的字符串对象。

但其他对象，如数组就是可变的对象。接下来我们会为大家介绍常用的 java.lang.String 方法。

创建字符串

创建字符串实例如下：

```
var greeting = "Hello World!";

或

var greeting:String = "Hello World!";
```

你不一定为字符串指定 String 类型，因为 Scala 编译器会自动推断出字符串的类型为 String。

当然我们也可以直接显示的声明字符串为 String 类型，如下实例：

```
object Test {
   val greeting: String = "Hello, World!"

   def main(args: Array[String]) {
      println( greeting )
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
Hello, world!
```

我们前面提到过 String 对象是不可变的，如果你需要创建一个可以修改的字符串，可以使用 String Builder 类，如下实例:

```
object Test {
   def main(args: Array[String]) {
      val buf = new StringBuilder;
      buf += 'a'
      buf ++= "bcdef"
      println( "buf is : " + buf.toString );
   }
}
```

运行实例？

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
buf is : abcdef
```

字符串长度

我们可以使用 length() 方法来获取字符串长度：

```
object Test {
   def main(args: Array[String]) {
      var palindrome = "www.runoob.com";
      var len = palindrome.length();
      println( "String Length is : " + len );
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
String Length is : 14
```

字符串连接

String 类中使用 concat() 方法来连接两个字符串：

```
string1.concat(string2);
```

实例演示：

```
scala> "菜鸟教程官网： ".concat("www.runoob.com");
res0: String = 菜鸟教程官网： www.runoob.com
```

同样你也可以使用加号(+)来连接：

```
scala> "菜鸟教程官网： " + " www.runoob.com"
res1: String = 菜鸟教程官网：  www.runoob.com
```

让我们看个完整实例:

```
object Test {
   def main(args: Array[String]) {
      var str1 = "菜鸟教程官网：";
      var str2 =  "www.runoob.com";
      var str3 =  "菜鸟教程的 Slogan 为：";
      var str4 =  "学的不仅是技术，更是梦想！";
      println( str1 + str2 );
      println( str3.concat(str4) );
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
菜鸟教程官网：www.runoob.com
菜鸟教程的 Slogan 为：学的不仅是技术，更是梦想！
```

创建格式化字符串

String 类中你可以使用 printf() 方法来格式化字符串并输出，String format() 方法可以返回 String 对象而不是 PrintStream 对象。以下实例演示了 printf() 方法的使用：

```
object Test {
   def main(args: Array[String]) {
      var floatVar = 12.456
      var intVar = 2000
      var stringVar = "菜鸟教程!"
      var fs = printf("浮点型变量为 " +
                  "%f, 整型变量为 %d, 字符串为 " +
                  " %s", floatVar, intVar, stringVar)
      println(fs)
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
浮点型变量为 12.456000, 整型变量为 2000, 字符串为  菜鸟教程!()
```

String 方法

下表列出了 java.lang.String 中常用的方法，你可以在 Scala 中使用：

方法	描述
char charAt(int index)	返回指定位置的字符
int compareTo(Object o)	比较字符串与对象
int compareTo(String anotherString)	按字典顺序比较两个字符串
int compareToIgnoreCase(String str)	按字典顺序比较两个字符串，不考虑大小写
String concat(String str)	将指定字符串连接到此字符串的结尾
boolean contentEquals(StringBuffer sb)	将此字符串与指定的 StringBuffer 比较。
static String copyValueOf(char[] data)	返回指定数组中表示该字符序列的 String
static String copyValueOf(char[] data, int offset, int count)	返回指定数组中表示该字符序列的 String
boolean endsWith(String suffix)	测试此字符串是否以指定的后缀结束
boolean equals(Object anObject)	将此字符串与指定的对象比较
boolean equalsIgnoreCase(String anotherString)	将此 String 与另一个 String 比较，不考虑大小写
byte getBytes()	使用平台的默认字符集将此 String 编码为 byte 序列，并将结果存储到一个新的 byte 数组中
byte[] getBytes(String charsetName	使用指定的字符集将此 String 编码为 byte 序列，并将结果存储到一个新的 byte 数组中
void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)	将字符从此字符串复制到目标字符数组
int hashCode()	返回此字符串的哈希码
int indexOf(int ch)	返回指定字符在此字符串中第一次出现处的索引
int indexOf(int ch, int fromIndex)	返返回在此字符串中第一次出现指定字符处的索引，从指定的索引开始搜索
int indexOf(String str)	返回指定子字符串在此字符串中第一次出现处的索引

int indexOf(String str, int fromIndex)	返回指定子字符串在此字符串中第一次出现处的索引，从指定的索引开始
String intern()	返回字符串对象的规范化表示形式
int lastIndexOf(int ch)	返回指定字符在此字符串中最后一次出现处的索引
int lastIndexOf(int ch, int fromIndex)	返回指定字符在此字符串中最后一次出现处的索引，从指定的索引处开始进行反向搜索
int lastIndexOf(String str)	返回指定子字符串在此字符串中最右边出现处的索引
int lastIndexOf(String str, int fromIndex)	返回指定子字符串在此字符串中最后一次出现处的索引，从指定的索引开始反向搜索
int length()	返回此字符串的长度
boolean matches(String regex)	告知此字符串是否匹配给定的正则表达式
boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)	测试两个字符串区域是否相等
boolean regionMatches(int toffset, String other, int ooffset, int len)	测试两个字符串区域是否相等
String replace(char oldChar, char newChar)	返回一个新的字符串，它是通过用 newChar 替换此字符串中出现的所有 oldChar 得到的
String replaceAll(String regex, String replacement	使用给定的 replacement 替换此字符串所有匹配给定的正则表达式的子字符串
String replaceFirst(String regex, String replacement)	使用给定的 replacement 替换此字符串匹配给定的正则表达式的第一个子字符串
String[] split(String regex)	根据给定正则表达式的匹配拆分此字符串
String[] split(String regex, int limit)	根据匹配给定的正则表达式来拆分此字符串
boolean startsWith(String prefix)	测试此字符串是否以指定的前缀开始
boolean startsWith(String prefix, int toffset)	测试此字符串从指定索引开始的子字符串是否以指定前缀开始。
CharSequence subSequence(int beginIndex, int endIndex)	返回一个新的字符序列，它是此序列的一个子序列
String substring(int beginIndex)	返回一个新的字符串，它是此字符串的一个子字符串
String substring(int beginIndex, int endIndex)	返回一个新字符串，它是此字符串的一个子字符串

char[] toCharArray()	将此字符串转换为一个新的字符数组
String toLowerCase()	使用默认语言环境的规则将此 String 中的所有字符都转换为小写
String toLowerCase(Locale locale)	使用给定 Locale 的规则将此 String 中的所有字符都转换为小写
String toString()	返回此对象本身（它已经是一个字符串！）
String toUpperCase()	使用默认语言环境的规则将此 String 中的所有字符都转换为大写
String toUpperCase(Locale locale)	使用给定 Locale 的规则将此 String 中的所有字符都转换为大写
String trim()	使用给定 Locale 的规则将此 String 中的所有字符都转换为大写
static String valueOf(primitive data type x)	返回指定类型参数的字符串表示形式

Scala 数组

Scala 语言中提供的数组是用来存储固定大小的同类型元素，数组对于每一门编辑应语言来说都是重要的数据结构之一。

声明数组变量并不是声明 number0、number1、...、number99 一个个单独的变量，而是声明一个就像 numbers 这样的变量，然后使用 numbers[0]、numbers[1]、...、numbers[99] 来表示一个个单独的变量。数组中某个指定的元素是通过索引来访问的。

数组的第一个元素索引为0，最后一个元素的索引为元素总数减1。

声明数组

以下是 Scala 数组声明的语法格式：

```
var z:Array[String] = new Array[String](3)

或

var z = new Array[String](3)
```

以上语法中，z 声明一个字符串类型的数组，数组长度为 3 ，可存储 3 个元素。我们可以为每个元素设置值，并通过索引来访问每个元素，如下所示：

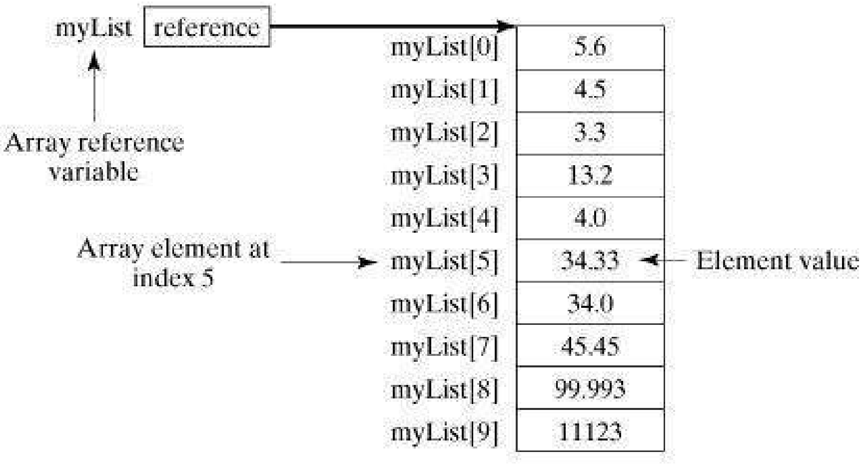
```
z(0) = "Runoob"; z(1) = "Baidu"; z(4/2) = "Google"
```

最后一个元素的索引使用了表达式 **4/2** 作为索引，类似于 **z(2) = "Google"**。

我们也可以使用以下方式来定义一个数组：

```
var z = Array("Runoob", "Baidu", "Google")
```

下图展示了一个长度为 10 的数组 myList，索引值为 0 到 9：



处理数组

数组的元素类型和数组的大小都是确定的，所以当处理数组元素时候，我们通常使用基本的 for 循环。

以下实例演示了数组的创建，初始化等处理过程：

```
object Test {
   def main(args: Array[String]) {
      var myList = Array(1.9, 2.9, 3.4, 3.5)

      // 输出所有数组元素
      for ( x <- myList ) {
         println( x )
      }

      // 计算数组所有元素的总会
      var total = 0.0;
      for ( i <- 0 to (myList.length - 1)) {
         total += myList(i);
      }
      println("总和为 " + total);

      // 查找数组中的最大元素
      var max = myList(0);
      for ( i <- 1 to (myList.length - 1) ) {
         if (myList(i) > max) max = myList(i);
      }
      println("最大值为 " + max);

   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
1.9
2.9
3.4
3.5
总和为 11.7
最大值为 3.5
```

多维数组

多维数组一个数组中的值可以是另一个数组，另一个数组的值也可以是一个数组。矩阵与表格是我们常见的二维数组。

以上是一个定义了二维数组的实例：

```
var myMatrix = ofDim[Int](3,3)
```

实例中数组中包含三个数组元素，每个数组元素又含有三个值。

接下来我们来看一个二维数组处理的完整实例：

```
import Array._

object Test {
   def main(args: Array[String]) {
      var myMatrix = ofDim[Int](3,3)

      // 创建矩阵
      for (i <- 0 to 2) {
         for ( j <- 0 to 2) {
            myMatrix(i)(j) = j;
         }
      }

      // 打印二维阵列
      for (i <- 0 to 2) {
         for ( j <- 0 to 2) {
            print(" " + myMatrix(i)(j));
         }
         println();
      }

   }
}
```

运行实例？

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
0 1 2
0 1 2
0 1 2
```

合并数组

以下实例中，我们使用 concat() 方法来合并两个数组，concat() 方法中接受多个数组参数：

```
import Array._

object Test {
   def main(args: Array[String]) {
      var myList1 = Array(1.9, 2.9, 3.4, 3.5)
      var myList2 = Array(8.9, 7.9, 0.4, 1.5)

      var myList3 =  concat( myList1, myList2)

      // 输出所有数组元素
      for ( x <- myList3 ) {
         println( x )
      }
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
1.9
2.9
3.4
3.5
8.9
7.9
0.4
1.5
```

创建区间数组

以下实例中，我们使用了 range() 方法来生成一个区间范围内的数组。range() 方法最后一个参数为步长，默认为 1：

```
import Array._

object Test {
   def main(args: Array[String]) {
      var myList1 = range(10, 20, 2)
      var myList2 = range(10,20)

      // 输出所有数组元素
      for ( x <- myList1 ) {
         print( " " + x )
      }
      println()
      for ( x <- myList2 ) {
         print( " " + x )
      }
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
10 12 14 16 18
10 11 12 13 14 15 16 17 18 19
```

Scala 数组方法

下表中为 Scala 语言中处理数组的重要方法，使用它前我们需要使用 **import Array._** 引入包。

方法	描述
def apply(x: T, xs: T*): Array[T]	创建指定对象 T 的数组, T 的值可以是 Unit, Double, Float, Long Boolean。
def concat[T](xss: Array[T]*): Array[T]	合并数组
def copy(src: AnyRef, srcPos: Int, dest: AnyRef, destPos: Int, length: Int): Unit	复制一个数组到另一个数组上。相等于 Java's System.arraycop destPos, length)。
def empty[T]: Array[T]	返回长度为 0 的数组
def iterate[T](start: T, len: Int)(f: (T) => T): Array[T]	返回指定长度数组，每个数组元素为指定函数的返回值。以上实为 3，计算函数为**a=>a+1**： `scala> Array.iterate(0,3)(a=>a+1) res1: Arr`
def fill[T](n: Int)(elem: => T): Array[T]	返回数组，长度为第一个参数指定，同时每个元素使用第二个参
def fill[T](n1: Int, n2: Int)(elem: => T): Array[Array[T]]	返回二数组，长度为第一个参数指定，同时每个元素使用第二个
def ofDim[T](n1: Int): Array[T]	创建指定长度的数组
def ofDim[T](n1: Int, n2: Int): Array[Array[T]]	创建二维数组
def ofDim[T](n1: Int, n2: Int, n3: Int): Array[Array[Array[T]]]	创建三维数组
def range(start: Int, end: Int, step: Int): Array[Int]	创建指定区间内的数组，step 为每个元素间的步长
def range(start: Int, end: Int): Array[Int]	创建指定区间内的数组
def tabulate[T](n: Int)(f: (Int)=> T): Array[T]	返回指定长度数组，每个数组元素为指定函数的返回值，默认从个元素： `scala> Array.tabulate(3)(a => a + 5) res0: Arra`
def tabulate[T](n1: Int, n2: Int)(f: (Int, Int) => T): Array[Array[T]]	返回指定长度的二维数组，每个数组元素为指定函数的返回值，

Scala Collection

Scala提供了一套很好的集合实现，提供了一些集合类型的抽象。

Scala 集合分为可变的和不可变的集合。

可变集合可以在适当的地方被更新或扩展。这意味着你可以修改，添加，移除一个集合的元素。

而不可变集合类，相比之下，永远不会改变。不过，你仍然可以模拟添加，移除或更新操作。但是这些操作将在每一种情况下都返回一个新的集合，同时使原来的集合不发生改变。

接下来我们将为大家介绍几种常用集合类型的应用：

序号	集合	描述
1	Scala List(列表)	List的特征是其元素以线性方式存储，集合中可以存放重复对象。 参考 API文档
2	Scala Set(集合)	Set是最简单的一种集合。集合中的对象不按特定的方式排序，并且没有重复对象。 参考 API文档
3	Scala Map(映射)	Map 是一种把键对象和值对象映射的集合，它的每一个元素都包含一对键对象和值对象。 参考 API文档
4	Scala 元组	元组是不同类型的值的集合
5	Scala Option	Option[T] 表示有可能包含值的容器，也可能不包含值。
6	Scala Iterator（迭代器）	迭代器不是一个容器，更确切的说是逐一访问容器内元素的方法。

实例

以下代码判断，演示了所有以上集合类型的定义实例：

```
// 定义整型 List
val x = List(1,2,3,4)

// 定义 Set
var x = Set(1,3,5,7)

// 定义 Map
val x = Map("one" -> 1, "two" -> 2, "three" -> 3)

// 创建两个不同类型元素的元组
val x = (10, "Runoob")

// 定义 Option
val x:Option[Int] = Some(5)
```

Scala List(列表)

Scala 列表类似于数组，它们所有元素的类型都相同，但是它们也有所不同：列表是不可变的，值一旦被定义了就不能改变，其次列表 具有递归的结构（也就是链接表结构）而数组不是。。

列表的元素类型 T 可以写成 List[T]。例如，以下列出了多种类型的列表：

```
// 字符串列表
val site: List[String] = List("Runoob", "Google", "Baidu")

// 整型列表
val nums: List[Int] = List(1, 2, 3, 4)

// 空列表
val empty: List[Nothing] = List()

// 二维列表
val dim: List[List[Int]] =
   List(
      List(1, 0, 0),
      List(0, 1, 0),
      List(0, 0, 1)
   )
```

构造列表的两个基本单位是 **Nil** 和 **::**

Nil 也可以表示为一个空列表。

以上实例我们可以写成如下所示：

```
// 字符串列表
val site = "Runoob" :: ("Google" :: ("Baidu" :: Nil))

// 整型列表
val nums = 1 :: (2 :: (3 :: (4 :: Nil)))

// 空列表
val empty = Nil

// 二维列表
val dim = (1 :: (0 :: (0 :: Nil))) ::
          (0 :: (1 :: (0 :: Nil))) ::
          (0 :: (0 :: (1 :: Nil))) :: Nil
```

列表基本操作

Scala列表有三个基本操作：

- `head` 返回列表第一个元素
- `tail` 返回一个列表，包含除了第一元素之外的其他元素
- `isEmpty` 在列表为空时返回true

对于Scala列表的任何操作都可以使用这三个基本操作来表达。实例如下:

```
object Test {
   def main(args: Array[String]) {
      val site = "Runoob" :: ("Google" :: ("Baidu" :: Nil))
      val nums = Nil

      println( "第一网站是 : " + site.head )
      println( "最后一个网站是 : " + site.tail )
      println( "查看列表 site 是否为空 : " + site.isEmpty )
      println( "查看 nums 是否为空 : " + nums.isEmpty )
   }
}
```

执行以上代码，输出结果为：

```
$ vim Test.scala
$ scala Test.scala
第一网站是 : Runoob
最后一个网站是 : List(Google, Baidu)
查看列表 site 是否为空 : false
查看 nums 是否为空 : true
```

连接列表

你可以使用 **:::** 运算符或 **List.:::()** 方法或 **List.concat()** 方法来连接两个或多个列表。实例如下:

```
object Test {
   def main(args: Array[String]) {
      val site1 = "Runoob" :: ("Google" :: ("Baidu" :: Nil))
      val site2 = "Facebook" :: ("Taobao" :: Nil)

      // 使用 ::: 运算符
      var fruit = site1 ::: site2
      println( "site1 ::: site2 : " + fruit )

      // 使用 Set.:::() 方法
      fruit = site1.:::(site2)
      println( "site1.:::(site2) : " + fruit )

      // 使用 concat 方法
      fruit = List.concat(site1, site2)
      println( "List.concat(site1, site2) : " + fruit  )


   }
}
```

执行以上代码，输出结果为：

```
$ vim Test.scala
$ scala Test.scala
site1 ::: site2 : List(Runoob, Google, Baidu, Facebook, Taobao)
site1.:::(site2) : List(Facebook, Taobao, Runoob, Google, Baidu)
List.concat(site1, site2) : List(Runoob, Google, Baidu, Facebook, Taobao)
```

List.fill()

我们可以使用 List.fill() 方法来创建一个指定重复数量的元素列表：

```
object Test {
   def main(args: Array[String]) {
      val site = List.fill(3)("Runoob") // 重复 Runoob 3次
      println( "site : " + site  )

      val num = List.fill(10)(2)         // 重复元素 2, 10 次
      println( "num : " + num  )
   }
}
```

执行以上代码，输出结果为：

```
$ vim Test.scala
$ scala Test.scala
site : List(Runoob, Runoob, Runoob)
num : List(2, 2, 2, 2, 2, 2, 2, 2, 2, 2)
```

List.tabulate()

List.tabulate() 方法是通过给定的函数来创建列表。

方法的第一个参数为元素的数量，可以是二维的，第二个参数为指定的函数，我们通过指定的函数计算结果并返回值插入到列表中，起始值为 0，实例如下：

```
object Test {
   def main(args: Array[String]) {
      // 通过给定的函数创建 5 个元素
      val squares = List.tabulate(6)(n => n * n)
      println( "一维 : " + squares  )

      // 创建二维列表
      val mul = List.tabulate( 4,5 )( _ * _ )
      println( "多维 : " + mul  )
   }
}
```

执行以上代码，输出结果为：

```
$ vim Test.scala
$ scala Test.scala
一维 : List(0, 1, 4, 9, 16, 25)
多维 : List(List(0, 0, 0, 0, 0), List(0, 1, 2, 3, 4), List(0, 2, 4, 6, 8), List(0, 3, 6, 9
```

List.reverse

List.reverse 用于将列表的顺序反转，实例如下：

```
object Test {
   def main(args: Array[String]) {
      val site = "Runoob" :: ("Google" :: ("Baidu" :: Nil))
      println( "site 反转前 : " + site )

      println( "site 反转前 : " + site.reverse )
   }
}
```

执行以上代码，输出结果为：

```
$ vim Test.scala
$ scala Test.scala
site 反转前 : List(Runoob, Google, Baidu)
site 反转前 : List(Baidu, Google, Runoob)
```

Scala List 常用方法

下表列出了 Scala List 常用的方法：

方法	描述
def +(elem: A): List[A]	为列表预添加元素
def ::(x: A): List[A]	在列表开头添加元素
def :::(prefix: List[A]): List[A]	在列表开头添加指定列表的元素
def ::(x: A): List[A]	在列表开头添加元素 x
def addString(b: StringBuilder): StringBuilder	将列表的所有元素添加到 StringBuilder
def addString(b: StringBuilder, sep: String): StringBuilder	将列表的所有元素添加到 StringBuilder，并指定分隔符
def apply(n: Int): A	通过列表索引获取元素
def contains(elem: Any): Boolean	检测列表中是否包含指定的元素
def copyToArray(xs:	

Array[A], start: Int, len: Int): Unit	
def distinct: List[A]	去除列表的重复元素，并返回新列表
def drop(n: Int): List[A]	丢弃前n个元素，并返回新列表
def dropRight(n: Int): List[A]	丢弃最后n个元素，并返回新列表
def dropWhile(p: (A) => Boolean): List[A]	从左向右丢弃元素，直到条件p不成立
def endsWith[B](that: Seq[B]): Boolean	检测列表是否以指定序列结尾
def equals(that: Any): Boolean	判断是否相等
def exists(p: (A) => Boolean): Boolean	判断列表中指定条件的元素是否存在。判断l是否存在某个元素: `scala> l.exists(s => s == "Hah") res7: Boolean = true`
def filter(p: (A) => Boolean): List[A]	输出符号指定条件的所有元素。过滤出长度为3的元素: `scala> l.filter(s => s.length == 3) res8: List[String] =`
def forall(p: (A) => Boolean): Boolean	检测所有元素。例如：判断所有元素是否以"H"开头： `scala> l.forall(s => s.startsWith("H")) res10: Boolean =`
def foreach(f: (A) => Unit): Unit	将函数应用到列表的所有元素
def head: A	获取列表的第一个元素
def indexOf(elem: A, from: Int): Int	从指定位置 from 开始查找元素第一次出现的位置
def init: List[A]	返回所有元素，除了最后一个
def intersect(that: Seq[A]): List[A]	计算多个集合的交集
def isEmpty: Boolean	检测列表是否为空
def iterator: Iterator[A]	创建一个新的迭代器来迭代元素
def last: A	返回最后一个元素
def lastIndexOf(elem:	在指定的位置 end 开始查找元素最后出现的位置

lastIndexOf(elem: A, end: Int): Int	在指定的位置 end 开始查找元素最后出现的位置
def length: Int	返回列表长度
def map[B](f: (A) => B): List[B]	通过给定的方法将所有元素重新计算
def max: A	查找最大元素
def min: A	查找最小元素
def mkString: String	列表所有元素作为字符串显示
def mkString(sep: String): String	使用分隔符将列表所有元素作为字符串显示
def reverse: List[A]	列表反转
def sorted[B >: A]: List[A]	列表排序
def startsWith[B](that: Seq[B], offset: Int): Boolean	检测列表在指定位置是否包含指定序列
def sum: A	计算集合元素之和
def tail: List[A]	返回所有元素，除了第一个
def take(n: Int): List[A]	提取列表的前n个元素
def takeRight(n: Int): List[A]	提取列表的后n个元素
def toArray: Array[A]	列表转换为数组
def toBuffer[B >: A]: Buffer[B]	返回缓冲区，包含了列表的所有元素
def toMap[T, U]: Map[T, U]	List 转换为 Map
def toSeq: Seq[A]	List 转换为 Seq
def toSet[B >: A]: Set[B]	List 转换为 Set
def toString(): String	列表转换为字符串

更多方法可以参考 API文档

Scala Set(集合)

Scala Set(集合)是没有重复的对象集合，所有的元素都是唯一的。

Scala 集合分为可变的和不可变的集合。

默认情况下，Scala 使用的是不可变集合，如果你想使用可变集合，需要引用 **scala.collection.mutable.Set** 包。

默认引用 scala.collection.immutable.Set，不可变集合实例如下：

```
val set = Set(1,2,3)
println(set.getClass.getName) //

println(set.exists(_ % 2 == 0)) //true
println(set.drop(1)) //Set(2,3)
```

如果需要使用可变集合需要引入 scala.collection.mutable.Set：

```
import scala.collection.mutable.Set // 可以在任何地方引入 可变集合

val mutableSet = Set(1,2,3)
println(mutableSet.getClass.getName) // scala.collection.mutable.HashSet

mutableSet.add(4)
mutableSet.remove(1)
mutableSet += 5
mutableSet -= 2

println(mutableSet) // Set(5, 3, 4)

val another = mutableSet.toSet
println(another.getClass.getName) // scala.collection.immutable.Set
```

> 注意： 虽然可变Set和不可变Set都有添加或删除元素的操作，但是有一个非常大的差别。对不可变Set进行操作，会产生一个新的set，原来的set并没有改变，这与List一样。而对可变Set进行操作，改变的是该Set本身，与ListBuffer类似。

集合基本操作

Scala集合有三个基本操作：

- `head` 返回集合第一个元素
- `tail` 返回一个集合，包含除了第一元素之外的其他元素
- `isEmpty` 在集合为空时返回true

对于Scala集合的任何操作都可以使用这三个基本操作来表达。实例如下:

```
object Test {
   def main(args: Array[String]) {
      val site = Set("Runoob", "Google", "Baidu")
      val nums: Set[Int] = Set()

      println( "第一网站是 : " + site.head )
      println( "最后一个网站是 : " + site.tail )
      println( "查看列表 site 是否为空 : " + site.isEmpty )
      println( "查看 nums 是否为空 : " + nums.isEmpty )
   }
}
```

执行以上代码，输出结果为：

```
$ vim Test.scala
$ scala Test.scala
第一网站是 : Runoob
最后一个网站是 : Set(Google, Baidu)
查看列表 site 是否为空 : false
查看 nums 是否为空 : true
```

连接集合

你可以使用 **++** 运算符或 **Set.++()** 方法来连接两个集合。如果元素有重复的就会移除重复的元素。实例如下：

```
object Test {
   def main(args: Array[String]) {
      val site1 = Set("Runoob", "Google", "Baidu")
      val site2 = Set("Faceboook", "Taobao")

      // ++ 作为运算符使用
      var site = site1 ++ site2
      println( "site1 ++ site2 : " + site )

      //  ++ 作为方法使用
      site = site1.++(site2)
      println( "site1.++(site2) : " + site )
   }
}
```

执行以上代码，输出结果为：

```
$ vim Test.scala
$ scala Test.scala
site1 ++ site2 : Set(Faceboook, Taobao, Google, Baidu, Runoob)
site1.++(site2) : Set(Faceboook, Taobao, Google, Baidu, Runoob)
```

查找集合中最大与最小元素

你可以使用 **Set.min** 方法来查找集合中的最小元素，使用 **Set.max** 方法查找集合中的最大元素。实例如下：

```
object Test {
   def main(args: Array[String]) {
      val num = Set(5,6,9,20,30,45)

      // 查找集合中最大与最小元素
      println( "Set(5,6,9,20,30,45) 集合中的最小元素是 : " + num.min )
      println( "Set(5,6,9,20,30,45) 集合中的最大元素是 : " + num.max )
   }
}
```

执行以上代码，输出结果为：

```
$ vim Test.scala
$ scala Test.scala
Set(5,6,9,20,30,45) 集合中的最小元素是 : 5
Set(5,6,9,20,30,45) 集合中的最大元素是 : 45
```

交集

你可以使用 **Set.&** 方法或 **Set.intersect** 方法来查看两个集合的交集元素。实例如下：

```
object Test {
   def main(args: Array[String]) {
      val num1 = Set(5,6,9,20,30,45)
      val num2 = Set(50,60,9,20,35,55)

      // 交集
      println( "num1.&(num2) : " + num1.&(num2) )
      println( "num1.intersect(num2) : " + num1.intersect(num2) )
   }
}
```

执行以上代码，输出结果为：

```
$ vim Test.scala
$ scala Test.scala
num1.&(num2) : Set(20, 9)
num1.intersect(num2) : Set(20, 9)
```

Scala Set 常用方法

下表列出了 Scala Set 常用的方法：

方法	描述
def +(elem: A): Set[A]	为集合添加新元素，x并创建一个新的集合，除非元素已存在
def -(elem: A): Set[A]	移除集合中的元素，并创建一个新的集合
def contains(elem: A): Boolean	如果元素在集合中存在，返回 true，否则返回 false。

def &(that: Set[A]): Set[A]	返回两个集合的交集
def &~(that: Set[A]): Set[A]	返回两个集合的差集
def +(elem1: A, elem2: A, elems: A*): Set[A]	通过添加传入指定集合的元素创建一个新的不可变集合
def ++(elems: A): Set[A]	合并两个集合
def -(elem1: A, elem2: A, elems: A*): Set[A]	通过移除传入指定集合的元素创建一个新的不可变集合
def addString(b: StringBuilder): StringBuilder	将不可变集合的所有元素添加到字符串缓冲区
def addString(b: StringBuilder, sep: String): StringBuilder	将不可变集合的所有元素添加到字符串缓冲区，并使用指定的分隔符
def apply(elem: A)	检测集合中是否包含指定元素
def count(p: (A) => Boolean): Int	计算满足指定条件的集合元素个数
def copyToArray(xs: Array[A], start: Int, len: Int): Unit	复制不可变集合元素到数组
def diff(that: Set[A]): Set[A]	比较两个集合的差集
def drop(n: Int): Set[A]]	返回丢弃前n个元素新集合
def dropRight(n: Int): Set[A]	返回丢弃最后n个元素新集合
def dropWhile(p: (A) => Boolean): Set[A]	从左向右丢弃元素，直到条件p不成立
def equals(that: Any): Boolean	equals 方法可用于任意序列。用于比较系列是否相等。
def exists(p: (A) => Boolean): Boolean	判断不可变集合中指定条件的元素是否存在。
def filter(p: (A) => Boolean): Set[A]	输出符合指定条件的所有不可变集合元素。
def find(p: (A) => Boolean): Option[A]	查找不可变集合中满足指定条件的第一个元素
def forall(p: (A) => Boolean): Boolean	查找不可变集合中满足指定条件的所有元素
def foreach(f: (A) => Unit): Unit	将函数应用到不可变集合的所有元素
def head: A	获取不可变集合的第一个元素
def init: Set[A]	返回所有元素，除了最后一个
def intersect(that: Set[A]): Set[A]	计算两个集合的交集
def isEmpty: Boolean	判断集合是否为空
def iterator: Iterator[A]	创建一个新的迭代器来迭代元素

def last: A	返回最后一个元素
def map[B](f: (A) => B): immutable.Set[B]	通过给定的方法将所有元素重新计算
def max: A	查找最大元素
def min: A	查找最小元素
def mkString: String	集合所有元素作为字符串显示
def mkString(sep: String): String	使用分隔符将集合所有元素作为字符串显示
def product: A	返回不可变集合中数字元素的积。
def size: Int	返回不可变集合元素的数量
def splitAt(n: Int): (Set[A], Set[A])	把不可变集合拆分为两个容器，第一个由前 n 个元素组成，第二个由剩下的元素组成
def subsetOf(that: Set[A]): Boolean	如果集合中含有子集返回 true，否则返回false
def sum: A	返回不可变集合中所有数字元素之和
def tail: Set[A]	返回一个不可变集合中除了第一元素之外的其他元素
def take(n: Int): Set[A]	返回前 n 个元素
def takeRight(n: Int):Set[A]	返回后 n 个元素
def toArray: Array[A]	将集合转换为数字
def toBuffer[B >: A]: Buffer[B]	返回缓冲区，包含了不可变集合的所有元素
def toList: List[A]	返回 List，包含了不可变集合的所有元素
def toMap[T, U]: Map[T, U]	返回 Map，包含了不可变集合的所有元素
def toSeq: Seq[A]	返回 Seq，包含了不可变集合的所有元素
def toString(): String	返回一个字符串，以对象来表示

更多方法可以参考 API文档

Scala Map(映射)

Map(映射)是一种可迭代的键值对（key/value）结构。

所有的值都可以通过键来获取。

Map 中的键都是唯一的。

Map 也叫哈希表（Hash tables）。

Map 有两种类型，可变与不可变，区别在于可变对象可以修改它，而不可变对象不可以。

默认情况下 Scala 使用不可变 Map。如果你需要使用可变集合，你需要显式的引入 **import scala.collection.mutable.Map** 类

在 Scala 中 你可以同时使用可变与不可变 Map，不可变的直接使用 Map，可变的使用 mutable.Map。以下实例演示了不可变 Map 的应用：

```
// 空哈希表，键为字符串，值为整型
var A:Map[Char,Int] = Map()

// Map 键值对演示
val colors = Map("red" -> "#FF0000", "azure" -> "#F0FFFF")
```

定义 Map 时，需要为键值对定义类型。如果需要添加 key-value 对，可以使用 + 号，如下所示：

```
A += ('I' -> 1)
A += ('J' -> 5)
A += ('K' -> 10)
A += ('L' -> 100)
```

Map 基本操作

Scala Map 有三个基本操作：

方法	描述
keys	返回 Map 所有的键(key)
values	返回 Map 所有的值(value)
isEmpty	在 Map 为空时返回true

实例

以下实例演示了以上三个方法的基本应用：

```
object Test {
   def main(args: Array[String]) {
      val colors = Map("red" -> "#FF0000",
                       "azure" -> "#F0FFFF",
                       "peru" -> "#CD853F")

      val nums: Map[Int, Int] = Map()

      println( "colors 中的键为 : " + colors.keys )
      println( "colors 中的值为 : " + colors.values )
      println( "检测 colors 是否为空 : " + colors.isEmpty )
      println( "检测 nums 是否为空 : " + nums.isEmpty )
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
colors 中的键为 : Set(red, azure, peru)
colors 中的值为 : MapLike(#FF0000, #F0FFFF, #CD853F)
检测 colors 是否为空 : false
检测 nums 是否为空 : true
```

Map 合并

你可以使用 **++** 运算符或 **Map.++()** 方法来连接两个 Map，Map 合并时会移除重复的 key。以下演示了两个 Map 合并的实例:

```
object Test {
   def main(args: Array[String]) {
      val colors1 = Map("red" -> "#FF0000",
                        "azure" -> "#F0FFFF",
                        "peru" -> "#CD853F")
      val colors2 = Map("blue" -> "#0033FF",
                        "yellow" -> "#FFFF00",
                        "red" -> "#FF0000")

      //  ++ 作为运算符
      var colors = colors1 ++ colors2
      println( "colors1 ++ colors2 : " + colors )

      //  ++ 作为方法
      colors = colors1.++(colors2)
      println( "colors1.++(colors2)) : " + colors )

   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
colors1 ++ colors2 : Map(blue -> #0033FF, azure -> #F0FFFF, peru -> #CD853F, yellow -> #F
colors1.++(colors2)) : Map(blue -> #0033FF, azure -> #F0FFFF, peru -> #CD853F, yellow ->
```

输出 Map 的 keys 和 values

以下通过 foreach 循环输出 Map 中的 keys 和 values：

```
object Test {
   def main(args: Array[String]) {
      val sites = Map("runoob" -> "http://www.runoob.com",
                      "baidu" -> "http://www.baidu.com",
                      "taobao" -> "http://www.taobao.com")

      sites.keys.foreach{ i =>
                          print( "Key = " + i )
                          println(" Value = " + sites(i) )}
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
Key = runoob Value = http://www.runoob.com
Key = baidu Value = http://www.baidu.com
Key = taobao Value = http://www.taobao.com
```

查看 Map 中是否存在指定的 Key

你可以使用 **Map.contains** 方法来查看 Map 中是否存在指定的 Key。实例如下：

```
object Test {
   def main(args: Array[String]) {
      val sites = Map("runoob" -> "http://www.runoob.com",
                      "baidu" -> "http://www.baidu.com",
                      "taobao" -> "http://www.taobao.com")

      if( sites.contains( "runoob" )){
           println("runoob 键存在，对应的值为 :"  + sites("runoob"))
      }else{
           println("runoob 键不存在")
      }
      if( sites.contains( "baidu" )){
           println("baidu 键存在，对应的值为 :"  + sites("baidu"))
      }else{
           println("baidu 键不存在")
      }
      if( sites.contains( "google" )){
           println("google 键存在，对应的值为 :"  + sites("google"))
      }else{
           println("google 键不存在")
      }
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
runoob 键存在，对应的值为 :http://www.runoob.com
baidu 键存在，对应的值为 :http://www.baidu.com
google 键不存在
```

Scala Map 方法

下表列出了 Scala Map 常用的方法：

方法	描述
def ++(xs: Map[(A, B)]): Map[A, B]	返回一个新的 Map，新的 Map xs 组成
def -(elem1: A, elem2: A, elems: A*): Map[A, B]	返回一个新的 Map, 移除 key 为 elem1, elem2 或其他 elems。
def --(xs: GTO[A]): Map[A, B]	返回一个新的 Map, 移除 xs 对象中对应的 key
def get(key: A): Option[B]	返回指定 key 的值
def iterator: Iterator[(A, B)]	创建新的迭代器，并输出 key/value 对
def addString(b: StringBuilder): StringBuilder	将 Map 中的所有元素附加到StringBuilder，可加入分隔符
def addString(b: StringBuilder, sep: String): StringBuilder	将 Map 中的所有元素附加到StringBuilder，可加入分隔符
def apply(key: A): B	返回指定键的值，如果不存在返回 Map 的默认方法

def clear(): Unit	清空 Map
def clone(): Map[A, B]	从一个 Map 复制到另一个 Map
def contains(key: A): Boolean	如果 Map 中存在指定 key，返回 true，否则返回 false。
def copyToArray(xs: Array[(A, B)]): Unit	复制集合到数组
def count(p: ((A, B)) => Boolean): Int	计算满足指定条件的集合元素数量
def default(key: A): B	定义 Map 的默认值，在 key 不存在时返回。
def drop(n: Int): Map[A, B]	返回丢弃前n个元素新集合
def dropRight(n: Int): Map[A, B]	返回丢弃最后n个元素新集合
def dropWhile(p: ((A, B)) => Boolean): Map[A, B]	从左向右丢弃元素，直到条件p不成立
def empty: Map[A, B]	返回相同类型的空 Map
def equals(that: Any): Boolean	如果两个 Map 相等(key/value 均相等)，返回true，否则返回false
def exists(p: ((A, B)) => Boolean): Boolean	判断集合中指定条件的元素是否存在
def filter(p: ((A, B))=> Boolean): Map[A, B]	返回满足指定条件的所有集合
def filterKeys(p: (A) => Boolean): Map[A, B]	返回符合指定条件的的不可变 Map
def find(p: ((A, B)) => Boolean): Option[(A, B)]	查找集合中满足指定条件的第一个元素
def foreach(f: ((A, B)) => Unit): Unit	将函数应用到集合的所有元素
def init: Map[A, B]	返回所有元素，除了最后一个
def isEmpty: Boolean	检测 Map 是否为空
def keys: Iterable[A]	返回所有的key/p>
def last: (A, B)	返回最后一个元素
def max: (A, B)	查找最大元素
def min: (A, B)	查找最小元素
def mkString: String	集合所有元素作为字符串显示
def product: (A, B)	返回集合中数字元素的积。
def remove(key: A): Option[B]	移除指定 key
def retain(p: (A, B) => Boolean): Map.this.type	如果符合满足条件的返回 true

def size: Int	返回 Map 元素的个数
def sum: (A, B)	返回集合中所有数字元素之和
def tail: Map[A, B]	返回一个集合中除了第一元素之外的其他元素
def take(n: Int): Map[A, B]	返回前 n 个元素
def takeRight(n: Int): Map[A, B]	返回后 n 个元素
def takeWhile(p: ((A, B)) => Boolean): Map[A, B]	返回满足指定条件的元素
def toArray: Array[(A, B)]	集合转数组
def toBuffer[B >: A]: Buffer[B]	返回缓冲区，包含了 Map 的所有元素
def toList: List[A]	返回 List，包含了 Map 的所有元素
def toSeq: Seq[A]	返回 Seq，包含了 Map 的所有元素
def toSet: Set[A]	返回 Set，包含了 Map 的所有元素
def toString(): String	返回字符串对象

更多方法可以参考 API文档

Scala 元组

与列表一样，元组也是不可变的，但与列表不同的是元组可以包含不同类型的元素。

元组的值是通过将单个的值包含在圆括号中构成的。例如：

```
val t = (1, 3.14, "Fred")
```

以上实例在元组中定义了三个元素，对应的类型分别为[Int, Double, java.lang.String]。

此外我们也可以使用以上方式来定义：

```
val t = new Tuple3(1, 3.14, "Fred")
```

元组的实际类型取决于它的元素的类型，比如 (99, "runoob") 是 Tuple2[Int, String]。 ('u', 'r', "the", 1, 4, "me") 为 Tuple6[Char, Char, String, Int, Int, String]。

目前 Scala 支持的元组最大长度为 22。对于更大长度你可以使用集合，或者扩展元组。

访问元组的元素可以通过数字索引，如下一个元组：

```
val t = (4,3,2,1)
```

我们可以使用 t._1 访问第一个元素， t._2 访问第二个元素，如下所示：

```
object Test {
   def main(args: Array[String]) {
      val t = (4,3,2,1)

      val sum = t._1 + t._2 + t._3 + t._4

      println( "元素之和为: " + sum )
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
元素之和为: 10
```

迭代元组

你可以使用 **Tuple.productIterator()** 方法来迭代输出元组的所有元素：

```
object Test {
   def main(args: Array[String]) {
      val t = (4,3,2,1)

      t.productIterator.foreach{ i =>println("Value = " + i )}
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
Value = 4
Value = 3
Value = 2
Value = 1
```

元组转为字符串

你可以使用 **Tuple.toString()** 方法将元组的所有元素组合成一个字符串，实例如下：

```
object Test {
   def main(args: Array[String]) {
      val t = new Tuple3(1, "hello", Console)

      println("连接后的字符串为: " + t.toString() )
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
连接后的字符串为: (1,hello,scala.Console$@4dd8dc3)
```

元素交换

你可以使用 **Tuple.swap** 方法来交换元组的元素。如下实例：

```
object Test {
   def main(args: Array[String]) {
      val t = new Tuple2("www.google.com", "www.runoob.com")

      println("交换后的元组: " + t.swap )
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
交换后的元组: (www.runoob.com,www.google.com)
```

Scala Option(选项)

Scala Option(选项)类型用来表示一个值是可选的（有值或无值)。

Option[T] 是一个类型为 T 的可选值的容器： 如果值存在， Option[T] 就是一个 Some[T] ，如果不存在， Option[T] 就是对象 None 。

接下来我们来看一段代码：

```
// 虽然 Scala 可以不定义变量的类型，不过为了清楚些，我还是
// 把他显示的定义上了

val myMap: Map[String, String] = Map("key1" -> "value")
val value1: Option[String] = myMap.get("key1")
val value2: Option[String] = myMap.get("key2")

println(value1) // Some("value1")
println(value2) // None
```

在上面的代码中，myMap 一个是一个 Key 的类型是 String，Value 的类型是 String 的 hash map，但不一样的是他的 get() 返回的是一个叫 Option[String] 的类别。

Scala 使用 Option[String] 来告诉你：「我会想办法回传一个 String，但也可能没有 String 给你」。

myMap 里并没有 key2 这笔数据，get() 方法返回 None。

Option 有两个子类别，一个是 Some，一个是 None，当他回传 Some 的时候，代表这个函式成功地给了你一个 String，而你可以透过 get() 这个函式拿到那个 String，如果他返回的是 None，则代表没有字符串可以给你。

另一个实例：

```
object Test {
   def main(args: Array[String]) {
      val sites = Map("runoob" -> "www.runoob.com", "google" -> "www.google.com")

      println("sites.get( \"runoob\" ) : " +  sites.get( "runoob" )) // Some(www.runoob.c
      println("sites.get( \"baidu\" ) : " +  sites.get( "baidu" ))  //  None
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
sites.get( "runoob" ) : Some(www.runoob.com)
sites.get( "baidu" ) : None
```

你也可以通过模式匹配来输出匹配值。实例如下：

```
object Test {
   def main(args: Array[String]) {
      val sites = Map("runoob" -> "www.runoob.com", "google" -> "www.google.com")

      println("show(sites.get( \"runoob\")) : " +
                                          show(sites.get( "runoob")) )
      println("show(sites.get( \"baidu\")) : " +
                                          show(sites.get( "baidu")) )
   }

   def show(x: Option[String]) = x match {
      case Some(s) => s
      case None => "?"
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
show(sites.get( "runoob")) : www.runoob.com
show(sites.get( "baidu")) : ?
```

getOrElse() 方法

你可以使用 getOrElse() 方法来获取元组中存在的元素或者使用其默认的值，实例如下：

```
object Test {
   def main(args: Array[String]) {
      val a:Option[Int] = Some(5)
      val b:Option[Int] = None

      println("a.getOrElse(0): " + a.getOrElse(0) )
      println("b.getOrElse(10): " + b.getOrElse(10) )
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
a.getOrElse(0): 5
b.getOrElse(10): 10
```

isEmpty() 方法

你可以使用 isEmpty() 方法来检测元组中的元素是否为 None，实例如下：

```
object Test {
   def main(args: Array[String]) {
      val a:Option[Int] = Some(5)
      val b:Option[Int] = None

      println("a.isEmpty: " + a.isEmpty )
      println("b.isEmpty: " + b.isEmpty )
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
a.isEmpty: false
b.isEmpty: true
```

Scala Option 常用方法

下表列出了 Scala Option 常用的方法：

<table>
<tr><th>方法</th><th>描述</th></tr>
<tr><td>def get: A</td><td>获取可选值</td></tr>
<tr><td>def isEmpty: Boolean</td><td>检测可选类型值是否为 None，是的话返回 true，否则返回 false</td></tr>
<tr><td>def productArity: Int</td><td>返回元素个数， A(x_1, ..., x_k), 返回 k</td></tr>
<tr><td>def productElement(n: Int): Any</td><td>获取指定的可选项，以 0 为起始。即 A(x1, ..., x_k), 返回 x(n+1)， 0 < n < k.</td></tr>
<tr><td>def exists(p: (A) => Boolean): Boolean</td><td>如果可选项中指定条件的元素是否存在且不为 None 返回 true，否则返回 false。</td></tr>
<tr><td>def filter(p: (A) => Boolean): Option[A]</td><td>如果选项包含有值，而且传递给 filter 的条件函数返回 true， filter 会返回 Some 实例。 否则，返回值为 None 。</td></tr>
<tr><td>def filterNot(p: (A) => Boolean): Option[A]</td><td>如果选项包含有值，而且传递给 filter 的条件函数返回 false， filter 会返回 Some 实例。 否则，返回值为 None 。</td></tr>
<tr><td>def flatMap[B](f: (A) | => Option[B]): Option[B]</td><td>如果选项包含有值，则传递给函数 f 处理后返回，否则返回 None</td></tr>
<tr><td>def foreach[U](f: (A) | => U): Unit</td><td>如果选项包含有值，则将每个值传递给函数 f， 否则不处理。</td></tr>
<tr><td>def getOrElse[B >: A](default: => B) |: B</td><td>如果选项包含有值，返回选项值，否则返回设定的默认值。</td></tr>
<tr><td>def isDefined: Boolean</td><td>如果可选值是 Some 的实例返回 true，否则返回 false。</td></tr>
<tr><td>def iterator: Iterator[A]</td><td>如果选项包含有值，迭代出可选值。如果可选值为空则返回空迭代器。</td></tr>
<tr><td>def map[B](f: (A) | => B): Option[B]</td><td>如果选项包含有值， 返回由函数 f 处理后的 Some，否则返回 None</td></tr>
<tr><td>def orElse[B >: A](alternative: => Option[B]) |: Option[B]</td><td>如果一个 Option 是 None， orElse 方法会返回传名参数的值，否则，就直接返回这个 Option。</td></tr>
<tr><td>def orNull</td><td>如果选项包含有值返回选项值，否则返回 null。</td></tr>
</table>

Scala Iterator（迭代器）

Scala Iterator（迭代器）不是一个集合，它是一种用于访问集合的方法。

迭代器 it 的两个基本操作是 **next** 和 **hasNext** 。

调用 **it.next()** 会返回迭代器的下一个元素，并且更新迭代器的状态。

调用 **it.hasNext()** 用于检测集合中是否还有元素。

让迭代器 it 逐个返回所有元素最简单的方法是使用 while 循环：

```
object Test {
   def main(args: Array[String]) {
      val it = Iterator("Baidu", "Google", "Runoob", "Taobao")

      while (it.hasNext){
         println(it.next())
      }
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
Baidu
Google
Runoob
Taobao
```

查找最大与最小元素

你可以使用 **it.min** 和 **it.max** 方法从迭代器中查找最大与最小元素，实例如下:

```
object Test {
   def main(args: Array[String]) {
      val ita = Iterator(20,40,2,50,69, 90)
      val itb = Iterator(20,40,2,50,69, 90)

      println("最大元素是：" + ita.max )
      println("最小元素是：" + itb.min )

   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
最大元素是：90
最小元素是：2
```

获取迭代器的长度

你可以使用 **it.size** | 或 **it.length** | 方法来查看迭代器中的元素个数。实例如下：

```
object Test {
   def main(args: Array[String]) {
      val ita = Iterator(20,40,2,50,69, 90)
      val itb = Iterator(20,40,2,50,69, 90)

      println("ita.size 的值: " + ita.size )
      println("itb.length 的值: " + itb.length )

   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
ita.size 的值: 6
itb.length 的值: 6
```

Scala Iterator 常用方法

下表列出了 Scala Iterator 常用的方法：

方法	描述
def hasNext: Boolean	如果还有可返回的元素，返回true。
def next(): A	返回迭代器的下一个元素，并且更新迭代器的状态
def ++(that: => Iterator[A]): Iterator[A]	合并两个迭代器
def ++[B >: A](that :=> GenTraversableOnce[B]) |: Iterator[B]	合并两个迭代器
def addString(b: StringBuilder): StringBuilder	添加一个字符串到 StringBuilder b
def addString(b: StringBuilder, sep: String): StringBuilder	添加一个字符串到 StringBuilder b，并指定分隔符

def buffered: BufferedIterator[A]	迭代器都转换成 BufferedIterator
def contains(elem: Any): Boolean	检测迭代器中是否包含指定元素
def copyToArray(xs: Array[A], start: Int, len: Int): Unit	将迭代器中选定的值传给数组
def count(p: (A) => Boolean): Int	返回迭代器元素中满足条件p的元素总数。
def drop(n: Int): Iterator[A]	返回丢弃前n个元素新集合
def dropWhile(p: (A) => Boolean): Iterator[A]	从左向右丢弃元素，直到条件p不成立
def duplicate: (Iterator[A], Iterator[A])	生成两个能分别返回迭代器所有元素的迭代器。
def exists(p: (A) => Boolean): Boolean	返回一个布尔值，指明迭代器元素中是否存在满足p的元素。
def filter(p: (A) => Boolean): Iterator[A]	返回一个新迭代器 ，指向迭代器元素中所有满足条件p的元素。
def filterNot(p: (A) => Boolean): Iterator[A]	返回一个迭代器，指向迭代器元素中不满足条件p的元素。
def find(p: (A) => Boolean): Option[A]	返回第一个满足p的元素或None。注意：如果找到满足条件的元素，迭代器会被置于该元素之后；如果没有找到，会被置于终点。
def flatMap[B](f: (A) | => GenTraversableOnce[B]): Iterator[B]	针对迭代器的序列中的每个元素应用函数f，并返回指向结果序列的迭代器。
def forall(p: (A) => Boolean): Boolean	返回一个布尔值，指明 it 所指元素是否都满足p。
def foreach(f: (A) => Unit): Unit	在迭代器返回的每个元素上执行指定的程序 f
def hasDefiniteSize: Boolean	如果迭代器的元素个数有限则返回true（缺省等同于isEmpty）
def indexOf(elem: B): Int	返回迭代器的元素中index等于x的第一个元素。注意：迭代器会越过这个元素。
def indexWhere(p: (A) => Boolean): Int	返回迭代器的元素中下标满足条件p的元素。注意：迭代器会越过这个元素。
def isEmpty: Boolean	检查it是否为空, 为空返回 true，否则返回false（与hasNext相反）。
def isTraversableAgain: Boolean	Tests whether this Iterator can be repeatedly traversed.

def length: Int	返回迭代器元素的数量。
def map[B](f: (A) | => B): Iterator[B]	将 it 中的每个元素传入函数 f 后的结果生成新的迭代器。
def max: A	返回迭代器迭代器元素中最大的元素。
def min: A	返回迭代器迭代器元素中最小的元素。
def mkString: String	将迭代器所有元素转换成字符串。
def mkString(sep: String): String	将迭代器所有元素转换成字符串，并指定分隔符。
def nonEmpty: Boolean	检查容器中是否包含元素（相当于 hasNext）。
def padTo(len: Int, elem: A): Iterator[A]	首先返回迭代器所有元素，追加拷贝 elem 直到长度达到 len。
def patch(from: Int, patchElems: Iterator[B], replaced: Int): Iterator[B]	返回一个新迭代器，其中自第 from 个元素开始的 replaced 个元素被迭代器所指元素替换。
def product: A	返回迭代器所指数值型元素的积。
def sameElements(that: Iterator[_]): Boolean	判断迭代器和指定的迭代器参数是否依次返回相同元素
def seq: Iterator[A]	返回集合的系列视图
def size: Int	返回迭代器的元素数量
def slice(from: Int, until: Int): Iterator[A]	返回一个新的迭代器，指向迭代器所指向的序列中从开始于第 from 个元素、结束于第 until 个元素的片段。
def sum: A	返回迭代器所指数值型元素的和
def take(n: Int): Iterator[A]	返回前 n 个元素的新迭代器。
def toArray: Array[A]	将迭代器指向的所有元素归入数组并返回。
def toBuffer: Buffer[B]	将迭代器指向的所有元素拷贝至缓冲区 Buffer。
def toIterable: Iterable[A]	Returns an Iterable containing all elements of this traversable or iterator. This will not terminate for infinite iterators.
def toIterator: Iterator[A]	把迭代器的所有元素归入一个Iterator容器并返回。
def toList: List[A]	把迭代器的所有元素归入列表并返回
def toMap[T, U]: Map[T, U]	将迭代器的所有键值对归入一个Map并返回。
def toSeq: Seq[A]	将代器的所有元素归入一个Seq容器并返回。
def toString(): String	将迭代器转换为字符串
def zip[B](that: Iterator[B]) |: Iterator[(A, B)]	返回一个新迭代器，指向分别由迭代器和指定的迭代器 that 元素一一对应而成的二元组序列

更多方法可以参考 API文档

Scala 类和对象

类是对象的抽象，而对象是类的具体实例。类是抽象的，不占用内存，而对象是具体的，占用存储空间。类是用于创建对象的蓝图，它是一个定义包括在特定类型的对象中的方法和变量的软件模板。

我们可以使用 new 关键字来创建类的对象，实例如下：

```
class Point(xc: Int, yc: Int) {
   var x: Int = xc
   var y: Int = yc

   def move(dx: Int, dy: Int) {
      x = x + dx
      y = y + dy
      println ("x 的坐标点: " + x);
      println ("y 的坐标点: " + y);
   }
}
```

Scala中的类不声明为public，一个Scala源文件中可以有多个类。

以上实例的类定义了两个变量 **x** 和 **y** ，一个方法：**move**，方法没有返回值。

Scala 的类定义可以有参数，称为类参数，如上面的 xc, yc，类参数在整个类中都可以访问。

接着我们可以使用 new 来实例化类，并访问类中的方法和变量：

```
import java.io._

class Point(xc: Int, yc: Int) {
   var x: Int = xc
   var y: Int = yc

   def move(dx: Int, dy: Int) {
      x = x + dx
      y = y + dy
      println ("x 的坐标点: " + x);
      println ("y 的坐标点: " + y);
   }
}

object Test {
   def main(args: Array[String]) {
      val pt = new Point(10, 20);

      // 移到一个新的位置
      pt.move(10, 10);
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
x 的坐标点: 20
y 的坐标点: 30
```

Scala 继承

Scala继承一个基类跟Java很相似，只多了两点限制：

- 1、重写方法需要override关键字
- 2、只有主构造函数才可以往基类的构造函数里写参数。

Scala 的副构造函数必须调用主构造函数或另一个构造函数，在 Scala 里主构造函数如同一道关卡，类的实例需要通过他来初始化。

接下来让我们来看哥实例：

```
class Point(xc: Int, yc: Int) {
   var x: Int = xc
   var y: Int = yc

   def move(dx: Int, dy: Int) {
      x = x + dx
      y = y + dy
      println ("x 的坐标点: " + x);
      println ("y 的坐标点: " + y);
   }
}

class Location(override val xc: Int, override val yc: Int,
   val zc :Int) extends Point(xc, yc){
   var z: Int = zc

   def move(dx: Int, dy: Int, dz: Int) {
      x = x + dx
      y = y + dy
      z = z + dz
      println ("x 的坐标点 : " + x);
      println ("y 的坐标点 : " + y);
      println ("z 的坐标点 : " + z);
   }
}
```

Scala 使用 extends 关键字来继承一个类。实例中 Location 类继承了 Point 类。Point 称为父类(基类)，Location 称为子类。

继承会继承父类的所有属性和方法，Scala 只允许继承一个父类。

实例如下：

```
import java.io._

class Point(val xc: Int, val yc: Int) {
   var x: Int = xc
   var y: Int = yc
   def move(dx: Int, dy: Int) {
      x = x + dx
      y = y + dy
      println ("x 的坐标点 : " + x);
      println ("y 的坐标点 : " + y);
   }
}

class Location(override val xc: Int, override val yc: Int,
   val zc :Int) extends Point(xc, yc){
   var z: Int = zc

   def move(dx: Int, dy: Int, dz: Int) {
      x = x + dx
      y = y + dy
      z = z + dz
      println ("x 的坐标点 : " + x);
      println ("y 的坐标点 : " + y);
      println ("z 的坐标点 : " + z);
   }
}

object Test {
   def main(args: Array[String]) {
      val loc = new Location(10, 20, 15);

      // 移到一个新的位置
      loc.move(10, 10, 5);
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
x 的坐标点 : 20
y 的坐标点 : 30
z 的坐标点 : 20
```

Scala 单例对象

在 Scala 中，是没有 static 这个东西的，但是它也为我们提供了单例模式的实现方法，那就是使用关键字 object。

Scala 中使用单例模式时，除了定义的类之外，还要定义一个同名的 object 对象，它和类的区别是，object对象不能带参数。

当单例对象与某个类共享同一个名称时，他被称作是这个类的伴生对象：companion object。你必须在同一个源文件里定义类和它的伴生对象。类被称为是这个单例对象的伴生类：companion class。类和它的伴生对象可以互相访问其私有成员。

单例对象实例

```
import java.io._

class Point(val xc: Int, val yc: Int) {
   var x: Int = xc
   var y: Int = yc
   def move(dx: Int, dy: Int) {
      x = x + dx
      y = y + dy
   }
}

object Test {
   def main(args: Array[String]) {
      val point = new Point(10, 20)
      printPoint

      def printPoint{
         println ("x 的坐标点 : " + point.x);
         println ("y 的坐标点 : " + point.y);
      }
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
x 的坐标点 : 10
y 的坐标点 : 20
```

伴生对象实例

```
/* 文件名：Marker.scala
 * author:菜鸟教程
 * url:www.runoob.com
 */

// 私有构造方法
class Marker private(val color:String) {

  println("创建" + this)

  override def toString(): String = "颜色标记："+ color

}

// 伴生对象，与类共享名字，可以访问类的私有属性和方法
object Marker{

    private val markers: Map[String, Marker] = Map(
      "red" -> new Marker("red"),
      "blue" -> new Marker("blue"),
      "green" -> new Marker("green")
    )

    def apply(color:String) = {
      if(markers.contains(color)) markers(color) else null
    }

    def getMarker(color:String) = {
      if(markers.contains(color)) markers(color) else null
    }
    def main(args: Array[String]) {
        println(Marker("red"))
        // 单例函数调用，省略了.(点)符号
        println(Marker getMarker "blue")
    }
}
```

执行以上代码，输出结果为：

```
$ scalac Marker.scala
$ scala Marker
创建颜色标记：red
创建颜色标记：blue
创建颜色标记：green
颜色标记：red
颜色标记：blue
```

Scala Trait(特征)

Scala Trait(特征) 相当于 Java 的接口，实际上它比接口还功能强大。

与接口不同的是，它还可以定义属性和方法的实现。

一般情况下Scala的类只能够继承单一父类，但是如果是 Trait(特征) 的话就可以继承多个，从结果来看就是实现了多重继承。

Trait(特征) 定义的方式与类类似，但它使用的关键字是 **trait**，如下所示：

```
trait Equal {
  def isEqual(x: Any): Boolean
  def isNotEqual(x: Any): Boolean = !isEqual(x)
}
```

以上Trait(特征)由两个方法组成：**isEqual** 和 **isNotEqual**。isEqual 方法没有定义方法的实现，isNotEqual定义了方法的实现。子类继承特征可以实现未被实现的方法。所以其实 Scala Trait(特征)更像 Java 的抽象类。

以下演示了特征的完整实例：

```
/* 文件名：Test.scala
 * author:菜鸟教程
 * url:www.runoob.com
 */
trait Equal {
  def isEqual(x: Any): Boolean
  def isNotEqual(x: Any): Boolean = !isEqual(x)
}

class Point(xc: Int, yc: Int) extends Equal {
  var x: Int = xc
  var y: Int = yc
  def isEqual(obj: Any) =
    obj.isInstanceOf[Point] &&
    obj.asInstanceOf[Point].x == x
}

object Test {
   def main(args: Array[String]) {
      val p1 = new Point(2, 3)
      val p2 = new Point(2, 4)
      val p3 = new Point(3, 3)

      println(p1.isNotEqual(p2))
      println(p1.isNotEqual(p3))
      println(p1.isNotEqual(2))
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
false
true
true
```

特征构造顺序

特征也可以有构造器，由字段的初始化和其他特征体中的语句构成。这些语句在任何混入该特征的对象在构造是都会被执行。

构造器的执行顺序：

- 调用超类的构造器；
- 特征构造器在超类构造器之后、类构造器之前执行；
- 特质由左到右被构造；
- 每个特征当中，父特质先被构造；
- 如果多个特征共有一个父特质，父特质不会被重复构造
- 所有特征被构造完毕，子类被构造。

构造器的顺序是类的线性化的反向。线性化是描述某个类型的所有超类型的一种技术规格。

Scala 模式匹配

Scala 提供了强大的模式匹配机制，应用也非常广泛。

一个模式匹配包含了一系列备选项，每个都开始于关键字 **case**。每个备选项都包含了一个模式及一到多个表达式。箭头符号 **=>** 隔开了模式和表达式。

以下是一个简单的整型值模式匹配实例：

```
object Test {
   def main(args: Array[String]) {
      println(matchTest(3))

   }
   def matchTest(x: Int): String = x match {
      case 1 => "one"
      case 2 => "two"
      case _ => "many"
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
many
```

match 对应 Java 里的 switch，但是写在选择器表达式之后。即：选择器 **match {备选项}**。

match 表达式通过以代码编写的先后次序尝试每个模式来完成计算，只要发现有一个匹配的case，剩下的case不会继续匹配。

接下来我们来看一个不同数据类型的模式匹配：

```
object Test {
   def main(args: Array[String]) {
      println(matchTest("two"))
      println(matchTest("test"))
      println(matchTest(1))
      println(matchTest(6))

   }
   def matchTest(x: Any): Any = x match {
      case 1 => "one"
      case "two" => 2
      case y: Int => "scala.Int"
      case _ => "many"
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
2
many
one
scala.Int
```

实例中第一个 case 对应整型数值 1，第二个 case 对应字符串值 two，第二个 case 对应字符串值 two，第三个 case 对应类型模式，用于判断传入的值是否为整型，相比使用isInstanceOf来判断类型，使用模式匹配更好。第四个 case 表示默认的全匹配备选项，即没有找到其他匹配时的匹配项，类似 switch 中的 default。

使用样例类

使用了case关键字的类定义就是就是样例类(case classes)，样例类是种特殊的类，经过优化以用于模式匹配。

以下是样例类的简单实例:

```
object Test {
   def main(args: Array[String]) {
       val alice = new Person("Alice", 25)
    val bob = new Person("Bob", 32)
       val charlie = new Person("Charlie", 32)

    for (person <- List(alice, bob, charlie)) {
        person match {
            case Person("Alice", 25) => println("Hi Alice!")
            case Person("Bob", 32) => println("Hi Bob!")
            case Person(name, age) =>
               println("Age: " + age + " year, name: " + name + "?")
         }
      }
   }
   // 样例类
   case class Person(name: String, age: Int)
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
Hi Alice!
Hi Bob!
Age: 32 year, name: Charlie?
```

在声明样例类时，下面的过程自动发生了：

- 构造器的每个参数都成为val，除非显式被声明为var，但是并不推荐这么做；
- 在伴生对象中提供了apply方法，所以可以不使用new关键字就可构建对象；
- 提供unapply方法使模式匹配可以工作；
- 生成toString、equals、hashCode和copy方法，除非显示给出这些方法的定义。

Scala 正则表达式

Scala 通过 scala.util.matching 包种的 **Regex** 类来支持正则表达式。以下实例演示了使用正则表达式查找单词 **Scala** :

```
import scala.util.matching.Regex

object Test {
   def main(args: Array[String]) {
      val pattern = "Scala".r
      val str = "Scala is Scalable and cool"

      println(pattern findFirstIn str)
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
Some(Scala)
```

实例中使用 String 类的 r() 方法构造了一个Regex对象。

然后使用 findFirstIn 方法找到首个匹配项。

如果需要查看所有的匹配项可以使用 findAllIn 方法。

你可以使用 mkString( ) 方法来连接正则表达式匹配结果的字符串，并可以使用管道(|)来设置不同的模式：

```
import scala.util.matching.Regex

object Test {
   def main(args: Array[String]) {
      val pattern = new Regex("(S|s)cala")  // 首字母可以是大写 S 或小写 s
      val str = "Scala is scalable and cool"

      println((pattern findAllIn str).mkString(","))   // 使用逗号 , 连接返回结果
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
Scala,scala
```

如果你需要将匹配的文本替换为指定的关键词，可以使用 **replaceFirstIn( )** 方法来替换第一个匹配项，使用 **replaceAllIn( )** 方法替换所有匹配项，实例如下:

```
object Test {
   def main(args: Array[String]) {
      val pattern = "(S|s)cala".r
      val str = "Scala is scalable and cool"

      println(pattern replaceFirstIn(str, "Java"))
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
Java is scalable and cool
```

正则表达式

Scala 的正则表达式继承了 Java 的语法规则，Java 则大部分使用了 Perl 语言的规则。

下表我们给出了常用的一些正则表达式规则：

表达式	匹配规则
^	匹配输入字符串开始的位置。
$	匹配输入字符串结尾的位置。
.	匹配除"\r\n"之外的任何单个字符。
[...]	字符集。匹配包含的任一字符。例如，"[abc]"匹配"plain"中的"a"。
[^...]	反向字符集。匹配未包含的任何字符。例如，"[^abc]"匹配"plain"中"p"，"l"，"i"，"n"。
\A	匹配输入字符串开始的位置（无多行支持）
\z	字符串结尾(类似$，但不受处理多行选项的影响)
\Z	字符串结尾或行尾(不受处理多行选项的影响)
re*	重复零次或更多次
re+	重复一次或更多次
re?	重复零次或一次
re{ n}	重复n次
re{ n,}	
re{ n, m}	重复n到m次
a|b	匹配 a 或者 b
(re)	匹配 re,并捕获文本到自动命名的组里

(?: re)	匹配 re,不捕获匹配的文本，也不给此分组分配组号
(?> re)	贪婪子表达式
\w	匹配字母或数字或下划线或汉字
\W	匹配任意不是字母，数字，下划线，汉字的字符
\s	匹配任意的空白符,相等于 [\t\n\r\f]
\S	匹配任意不是空白符的字符
\d	匹配数字，类似 [0-9]
\D	匹配任意非数字的字符
\G	当前搜索的开头
\n	换行符
\b	通常是单词分界位置，但如果在字符类里使用代表退格
\B	匹配不是单词开头或结束的位置
\t	制表符
\Q	开始引号：**\Q(a+b)*3\E** 可匹配文本 "(a+b)*3"。
\E	结束引号：**\Q(a+b)*3\E** 可匹配文本 "(a+b)*3"。

正则表达式实例

实例	描述
.	匹配除"\r\n"之外的任何单个字符。
[Rr]uby	匹配 "Ruby" 或 "ruby"
rub[ye]	匹配 "ruby" 或 "rube"
[aeiou]	匹配小写字母 ：aeiou
[0-9]	匹配任何数字，类似 [0123456789]
[a-z]	匹配任何 ASCII 小写字母
[A-Z]	匹配任何 ASCII 大写字母
[a-zA-Z0-9]	匹配数字，大小写字母
[^aeiou]	匹配除了 aeiou 其他字符
[^0-9]	匹配除了数字的其他字符
\d	匹配数字，类似: [0-9]
\D	匹配非数字，类似: [^0-9]
\s	匹配空格，类似: [\t\r\n\f]
\S	匹配非空格，类似: [^ \t\r\n\f]
\w	匹配字母，数字，下划线，类似: [A-Za-z0-9_]
\W	匹配非字母，数字，下划线，类似: [^A-Za-z0-9_]
ruby?	匹配 "rub" 或 "ruby": y 是可选的
ruby*	匹配 "rub" 加上 0 个或多个的 y。
ruby+	匹配 "rub" 加上 1 个或多个的 y。
\d{3}	刚好匹配 3 个数字。
\d{3,}	匹配 3 个或多个数字。
\d{3,5}	匹配 3 个、4 个或 5 个数字。
\D\d+	无分组 ： + 重复 \d
(\D\d)+/	分组 ： + 重复 \D\d 对
([Rr]uby(,)?)+	匹配 "Ruby"、"Ruby, ruby, ruby"，等等

注意上表中的每个字符使用了两个反斜线。这是因为在 Java 和 Scala 中字符串中的反斜线是转义字符。所以如果你要输出 ..，你需要在字符串中写成 .\. 来获取一个反斜线。查看以下实例：

```
import scala.util.matching.Regex

object Test {
   def main(args: Array[String]) {
      val pattern = new Regex("abl[ae]\\d+")
      val str = "ablaw is able1 and cool"

      println((pattern findAllIn str).mkString(","))
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
able1
```

Scala 异常处理

Scala 的异常处理和其它语言比如 Java 类似。

Scala 的方法可以通过抛出异常的方法的方式来终止相关代码的运行，不必通过返回值。

抛出异常

Scala 抛出异常的方法和 Java一样，使用 throw 方法，例如，抛出一个新的参数异常：

```
throw new IllegalArgumentException
```

捕获异常

异常捕捉的机制与其他语言中一样，如果有异常发生，catch字句是按次序捕捉的。因此，在catch字句中，越具体的异常越要靠前，越普遍的异常越靠后。 如果抛出的异常不在catch字句中，该异常则无法处理，会被升级到调用者处。

捕捉异常的catch子句，语法与其他语言中不太一样。在Scala里，借用了模式匹配的思想来做异常的匹配，因此，在catch的代码里，是一系列case字句，如下例所示：

```
import java.io.FileReader
import java.io.FileNotFoundException
import java.io.IOException

object Test {
   def main(args: Array[String]) {
      try {
         val f = new FileReader("input.txt")
      } catch {
         case ex: FileNotFoundException =>{
            println("Missing file exception")
         }
         case ex: IOException => {
            println("IO Exception")
         }
      }
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
Missing file exception
```

catch字句里的内容跟match里的case是完全一样的。由于异常捕捉是按次序，如果最普遍的异常，Throwable，写在最前面，则在它后面的case都捕捉不到，因此需要将它写在最后面。

finally 语句

finally 语句用于执行不管是正常处理还是有异常发生时都需要执行的步骤，实例如下：

```
import java.io.FileReader
import java.io.FileNotFoundException
import java.io.IOException

object Test {
   def main(args: Array[String]) {
      try {
         val f = new FileReader("input.txt")
      } catch {
         case ex: FileNotFoundException => {
            println("Missing file exception")
         }
         case ex: IOException => {
            println("IO Exception")
         }
      } finally {
         println("Exiting finally...")
      }
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
Missing file exception
Exiting finally...
```

Scala 提取器(Extractor)

提取器是从传递给它的对象中提取出构造该对象的参数。

Scala 标准库包含了一些预定义的提取器，我们会大致的了解一下它们。

Scala 提取器是一个带有unapply方法的对象。unapply方法算是apply方法的反向操作：unapply接受一个对象，然后从对象中提取值，提取的值通常是用来构造该对象的值。

以下实例演示了邮件地址的提取器对象：

```
object Test {
   def main(args: Array[String]) {

      println ("Apply 方法 : " + apply("Zara", "gmail.com"));
      println ("Unapply 方法 : " + unapply("Zara@gmail.com"));
      println ("Unapply 方法 : " + unapply("Zara Ali"));

   }
   // 注入方法 (可选)
   def apply(user: String, domain: String) = {
      user +"@"+ domain
   }

   // 提取方法（必选）
   def unapply(str: String): Option[(String, String)] = {
      val parts = str split "@"
      if (parts.length == 2){
         Some(parts(0), parts(1))
      }else{
         None
      }
   }
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
Apply 方法 : Zara@gmail.com
Unapply 方法 : Some((Zara,gmail.com))
Unapply 方法 : None
```

以上对象定义了两个方法： **apply** 和 **unapply** 方法。通过 apply 方法我们无需使用 new 操作就可以创建对象。所以你可以通过语句 Test("Zara", "gmail.com") 来构造一个字符串 "Zara@gmail.com"。

unapply方法算是apply方法的反向操作：unapply接受一个对象，然后从对象中提取值，提取的值通常是用来构造该对象的值。实例中我们使用 Unapply 方法从对象中提取用户名和邮件地址的后缀。

实例中 unapply 方法在传入的字符串不是邮箱地址时返回 None。代码演示如下：

```
unapply("Zara@gmail.com") 相等于 Some("Zara", "gmail.com")
unapply("Zara Ali") 相等于 None
```

提取器使用模式匹配

在我们实例化一个类的时，可以带上0个或者多个的参数，编译器在实例化的时会调用 apply 方法。我们可以在类和对象中都定义 apply 方法。

就像我们之前提到过的，unapply 用于提取我们指定查找的值，它与 apply 的操作相反。 当我们在提取器对象中使用 match 语句是，unapply 将自动执行，如下所示：

```
object Test {
   def main(args: Array[String]) {

      val x = Test(5)
      println(x)

      x match
      {
         case Test(num) => println(x + " 是 " + num + " 的两倍！")
         //unapply 被调用
         case _ => println("无法计算")
      }

   }
   def apply(x: Int) = x*2
   def unapply(z: Int): Option[Int] = if (z%2==0) Some(z/2) else None
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
$ scala Test
10
10 是 5 的两倍！
```

Scala 文件 I/O

Scala 进行文件写操作，直接用的都是 java中 的 I/O 类 （**java.io.File**）：

```
import java.io._

object Test {
   def main(args: Array[String]) {
      val writer = new PrintWriter(new File("test.txt" ))

      writer.write("菜鸟教程")
      writer.close()
   }
}
```

执行以上代码，会在你的当前目录下生产一个 test.txt 文件，文件内容为"菜鸟教程":

```
$ scalac Test.scala
$ scala Test
$ cat test.txt
菜鸟教程
```

从屏幕上读取用户输入

有时候我们需要接收用户在屏幕输入的指令来处理程序。实例如下：

```
object Test {
   def main(args: Array[String]) {
      print("请输入菜鸟教程官网 : " )
      val line = Console.readLine

      println("谢谢，你输入的是: " + line)
   }
}
```

执行以上代码，屏幕上会显示如下信息:

```
$ scalac Test.scala
$ scala Test
请输入菜鸟教程官网 : www.runoob.com
谢谢，你输入的是: www.runoob.com
```

从文件上读取内容

从文件读取内容非常简单。我们可以使用 Scala 的 **Source** 类及伴生对象来读取文件。以下实例演示了从 "test.txt"(之前已创建过) 文件中读取内容:

```
import scala.io.Source

object Test {
   def main(args: Array[String]) {
      println("文件内容为:" )

      Source.fromFile("test.txt" ).foreach{
         print
      }
   }
}
```

执行以上代码，输出结果为:

```
$ scalac Test.scala
$ scala Test
文件内容为:
菜鸟教程
```